POLITECNICO DI TORINO

Master of Science in Electronic Engineering

Master Thesis

Speed Enhancement Methods for HEVC Interpolation Filters

Supervisors:

Prof. Maurizio Martina

Prof. Guido Masera

Candidate:

Stefania Preatto

September, 2018

Abstract

Video sequences are largely diffused and have several applications worldwide. As a matter of fact, the necessity for a proficient video coding standard is required, by replacing the HM reference model with the HEVC project. The interpolations filters of this system represent the bottleneck for the CPU time: therefore an hardware implementation is strictly necessary in order to fulfill a short time coding requirement.
The purpose of this work of thesis is to find an appropriate internal architecture for the adders that are involved in the filtering operation, in order to further increase the throughput of the system. After a first analysis on input data statistics, several techniques are applied to enhance the efficiency of the entire system. Both exact and approximate adders are introduced, with a further examination on the impact of these choices on the performances of the reference system.
In the following, Ch.1 aims to give a general introduction of video coding, HEVC interpolation filters and to present the starting hardware reference structure. In Ch.2 parallel-prefix adders topologies are introduced to solve the time efficiency issue for the Chroma Legacy architecture, while Ch.3 explores the adoption of adaptive approximate computing with Generic Accuracy Configurable adders for the Luma Legacy architecture. Finally in Ch.4 the different adders topologies, presented in the previous sections, are applied to the approximate DCT-IF architectures in order to enhance performances.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Prof. Maurizio Martina for the continuous support of my work, for his patience, motivation and advice he has provided me through these months. Besides being an admirable teacher, he is a man of extreme kindness and availability who is always prepared to help and listen to his students. I would also like to thank Prof. Maurizio Masera for his useful and constructive teaching which allows me to have solid basis and method to face new issues in projects and work.
Thanks to people who stand by me, your sustain was worth more than I can express on paper. Heartfelt thanks go to Phil, for supporting and strengthening me both in university and in everyday experiences. Special mention to my closest friends Elvio, Chiara, Damiano, Francesca, Monica and Erica, for standing next to me during this journey: there is no distance that can separate a true friendship.
Last but not the least, I wish to thank my family, my parents Marzia and Roberto and my brother Luca, for always supporting and encouraging me throughout my life. Thanks to my dad, for imparting me passion and dedication to work and electronics. Finally I am particularly grateful to my grandparents, Nino and Renata: this experience would not have been possible without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to HEVC Interpolation Filters

High resolution video is expanding very rapidly in the last few years. As a consequence, a fast coding process is required to achieve suitable results for real time encoding, by replacing the HM reference model with the HEVC project. The interpolations filters of this system represent the bottleneck for the CPU time: therefore an hardware implementation is strictly necessary in order to fulfill a short time coding requirement.
This chapter aims to give a general introduction of video coding and HEVC interpolation filters and to present the starting hardware reference structure to improve the performances of the entire system.

1.1 Video Compression

Video sequences are largely diffused and have several applications worldwide. As a matter of fact, the necessity for a proficient video coding standard is required.
Since objects tend to move between consecutive frames, block-based motion compensation is the mostly adopted technique to take this movement into account. Each frame is divided into blocks of pixels and for every block to be predicted the encoder looks for the closest match in the reference frame.

1.1.1 Video Representation

The evolution of video representation from black-and-white to color television (RGB) implies the problem of backward compatibility, which was faced with the creation of a composite color signal(YUV). This last one is composed of three different factors:

- a luminance component Y that correlates to black and white television:

$$Y = 0.299R + 0.587G + 0.114B$$

  with R, G, B, respectively red, green and blue components

- two chrominance signals C_b(U) and C_r(V):

$$C_b = B - Y$$
$$C_r = R - Y$$

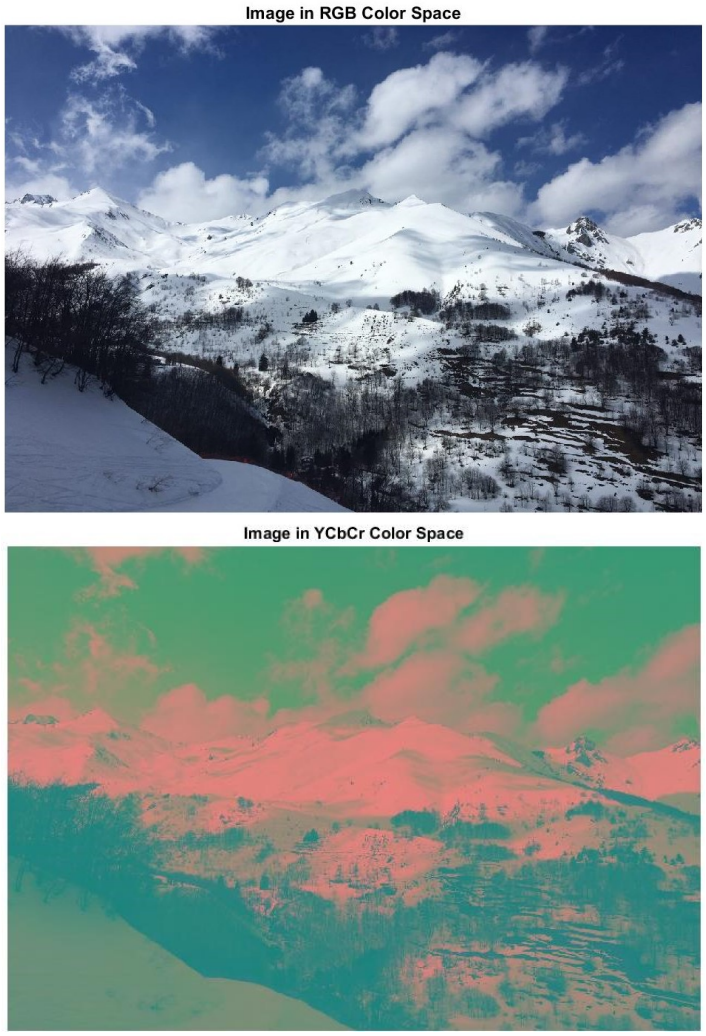

Figure 1.1: RGB and YUV images of Mountains, 1280x894

1.1.2 Video Structure

A modern video sequence is based on a similar idea to the one introduced by the MPEG-1 video standard [1] that listed three different frame types:

- *I frames*: coded with no reference to past frames. Indeed it's important to notice that sometimes it's possible to decode a sequence starting from the first frame, while in other cases it's not necessary to access the video sequence from the very beginning but from a random point;
- *B frames* (Bidirectionally predictive coded): introduced for prediction of both past and future frames;
- *P frames* (Predictive coded): used to forecast previous coded pictures exploiting motion compensation prediction.

B and P slices are introduced to improve the compression efficiency. Different frames are included in a group of picture (GOP) that is the smallest random access unit in the video sequence.

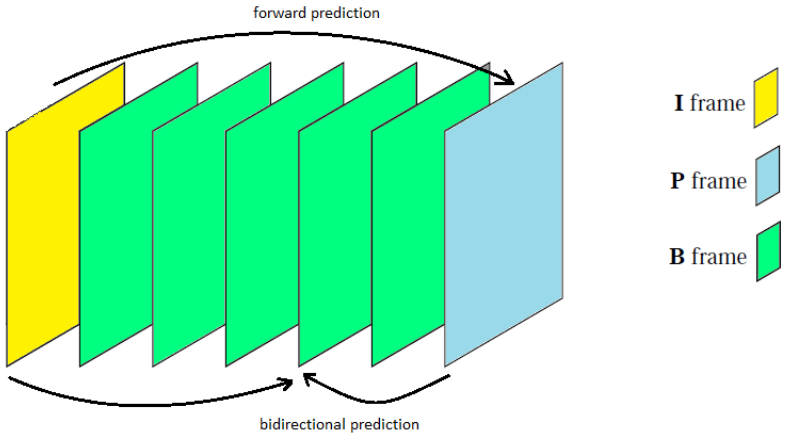

Figure 1.2: Group of pictures with I, B, P frames

1.2 High Efficiency Video Coding

Over the years video coding standards have evolved as a result of the development of two renowned standards: ITU-T, which presented the H.261 and H.263, and ISO/IEC, that produced the MPEG-1,MPEG-4, H.262/MPEG-2, H.264/MPEG-4 AVC. This this last one is the main predecessor of HEVC project.
High Efficiency Video Coding has been introduced to fix the raised use of parallel processing architectures and the increased video resolution, which is required in many applications targeting different devices (e.g. mobile phones, tablets or PCs).

1.2.1 HEVC Structure and Coding

Let's analyze the HEVC execution flow. Each input image is divided into pixel blocks and the prediction generated by means of the previous processed frames (that are stored within the Decoded Prediction Buffer) is subtracted each time from the new input frame; the output of this operation passes through transformation and quantization before being processed with entropy coding and given as output bit stream. Here a list of the main features of the HEVC design:

- **Transformation**: A two-dimensional Discrete Cosine Transform (DCT) is used executing a 1D transformation first on rows and then on columns. The prediction residual (out of the subtraction) is coded using two block transforms: a DCT and a DST (Discrete Sine Transform). The latter is necessary as inverse transformation, together with the inverse quantization, to provide the decoded frame to the motion compensation;
- **Quantization**: required to deal with the wide variation of values out of the DCT. The Quantization Parameter (QP) is fundamental in determining the block partitioning, since the lower the QP, the smaller the variations to be captured, so the higher the required bitrate;
- **Intrapicture prediction** (Intrapicture Estimation and Intrapicture Compensation): used in order to code the first frame and every random access point in a video sequence. Its main aim is to exploit the spatial dependency between two neighboring blocks,

supporting directional, planar and DC prediction modes. It uses some prediction from region-to-region in the same picture without any dependence on the other pictures;

- **Interpicture prediction**: is applied to all the other frames. A motion vector (MV) is chosen to predict the samples of each block; the employment of motion compensation will lead the encoder and the decoder to generate identical interpicture prediction signals;
- **Motion Estimation** (ME): given as input the new frame to be encoded and the previous reconstructed frames, it creates the Motion Vector which is sent to the Motion Compensation and to the entropy coding;
- **Motion Compensation** (MC): generates the prediction pixel block. While Motion Vector exploits quarter-sample precision, fractional-sample positions are interpolated through 7-tap or 8-tap filters;
- **Entropy Coding**: based on Context Adaptive Binary Arithmetic Coding (CABAC). The basic idea is to determine a separate probability model for each symbol that is coded as a bin; this is processed by dividing a range of representation in two subranges and selecting the one where the symbol is as a new range.

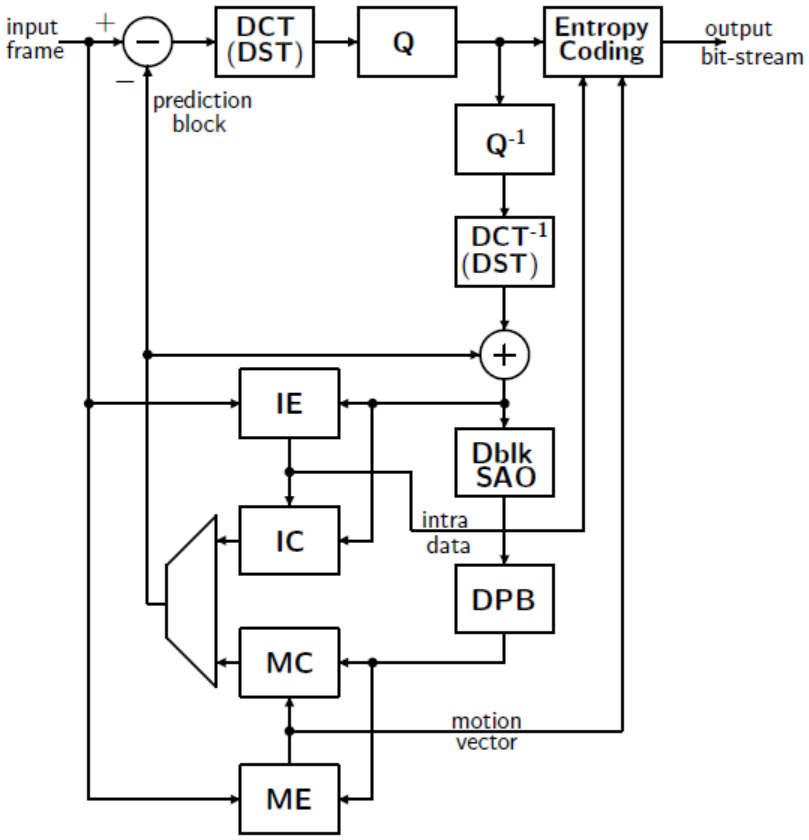

Figure 1.3: HEVC Block Diagram, figure courtesy of A.Giannini [2]

1.2.2 Software timing analysis

Since HEVC is intended as the newest video coding standard that is designed to achieve coding efficiency and data loss flexibility, a timing and complexity analysis should be carried out. With the HM reference software for HEVC [6], as presented in [2], the encoding time reaches more than ten hours. Even if this is not intended to provide a real model of an HEVC encoder, it's useful to provide some important informations on the most time consuming components. The obtained results are shown in figures 1.4 and 1.5.

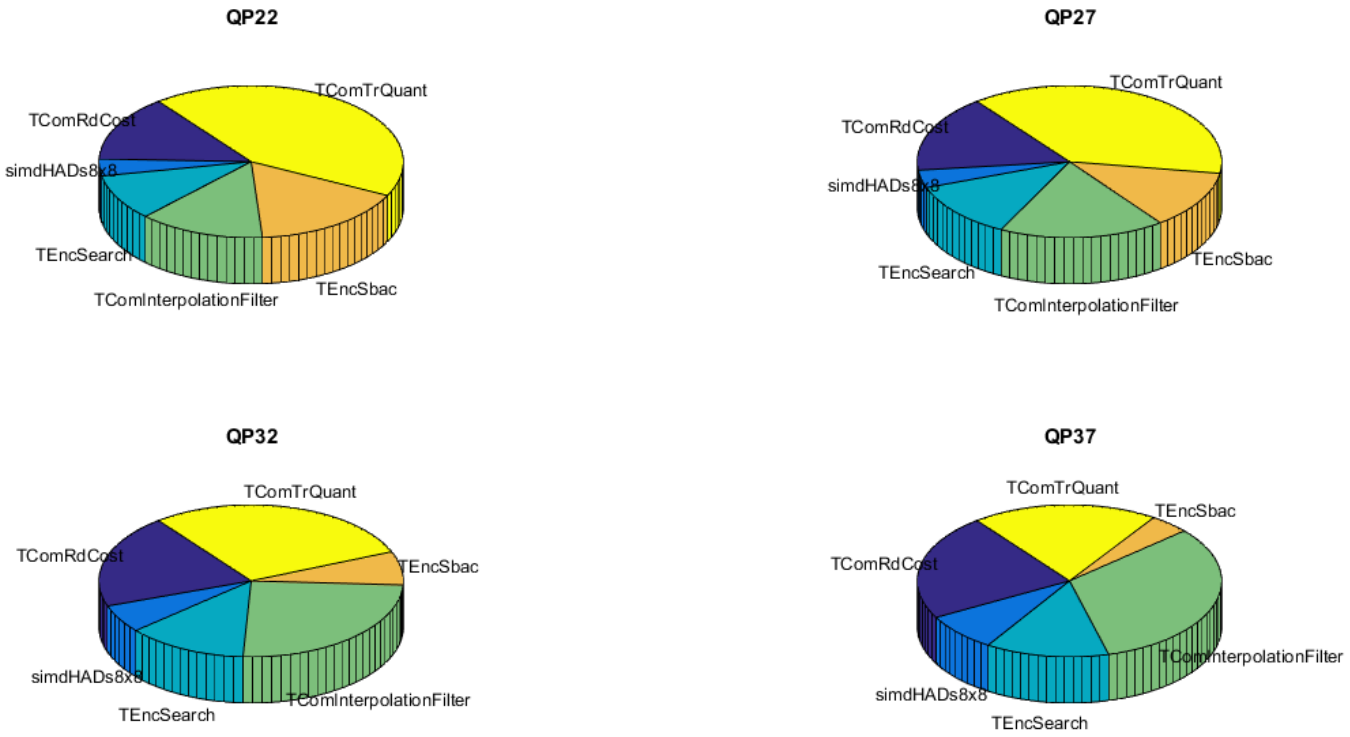

Figure 1.4: HM16.15 [6] Encoding time results from Software Profiling [2]

As far as simulations are concerned, it comes out that the interpolation filters seem to be the bottleneck for both the encoding and the decoding processes: indeed the percentage of CPU time spent on the interpolation filters increases with the Quantization Parameter. They are employed for two main aims: to enhance the encoding quality in Motion Compensation and to recover the predicted block from the Motion Vector and the already processed frames in Motion Estimation. Therefore it seems strongly suggested to provide an hardware implementation for the proposed HEVC interpolation filters in order to obtain an acceptable real-time encoding.

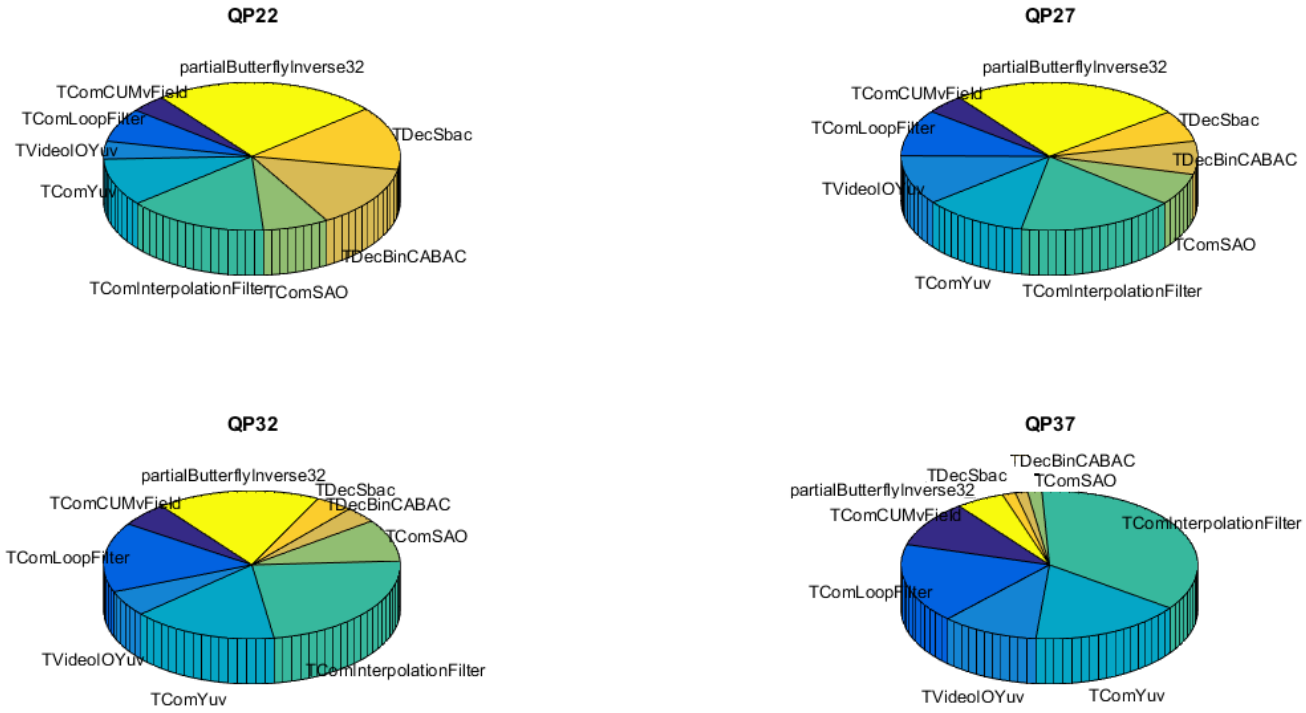

Figure 1.5: HM16.15 [6] Decoding time results from Software Profiling [2]

1.3 Fractional Sample Interpolation

As reported in the previous section, interpolation filters represent the most expensive computational block at both decoder and encoder's side when a high QP is required. The analysis and the implementation of an hardware accelerator for fractional sample interpolation is presented in [2].

1.3.1 Filter Design

A single separable interpolation process for all fractional pixels is applied in HEVC: this will lead to a reduced error because just one rounding process is applied with respect to its predecessor H.264/AVC algorithm. Different order filters are applied:

- *Luma Interpolation*: an eight-tap filter is used for pixels at half-samples position, while a seven-tap is employed for quarter-sample pixels. Coefficients are reported in table 1.1.

	-3	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**
hfilter[i]	-1	4	-11	40	40	-11	4	-1
qfilter[i]	-1	4	-10	58	17	-5	1	-

Table 1.1: Filter coefficients for luma fractional sample interpolation [3]

- *Chroma Interpolation*: since this signal is smoother with respect to the Y component, four four-tap filters are sufficient to accomplish the interpolation. Coefficients are listed in table 1.2. These last ones are used to interpolate from the $1/8^{th}$ to the $4/8^{th}$ fractional position; by mirroring values for filter3[1-i], filter2[1-i] and filter1[1-i], the $5/8^{th}$, $6/8^{th}$ and $7/8^{th}$ fractional positions are obtained by symmetry.

	-1	**0**	**1**	**2**
filter1[i] (1/8)	-2	58	10	-2
filter2[i] (2/8)	-4	54	16	-2
filter3[i] (3/8)	-6	46	28	-4
filter4[i] (4/8)	-4	36	36	-4

Table 1.2: Filter coefficients for chroma fractional sample interpolation [3]

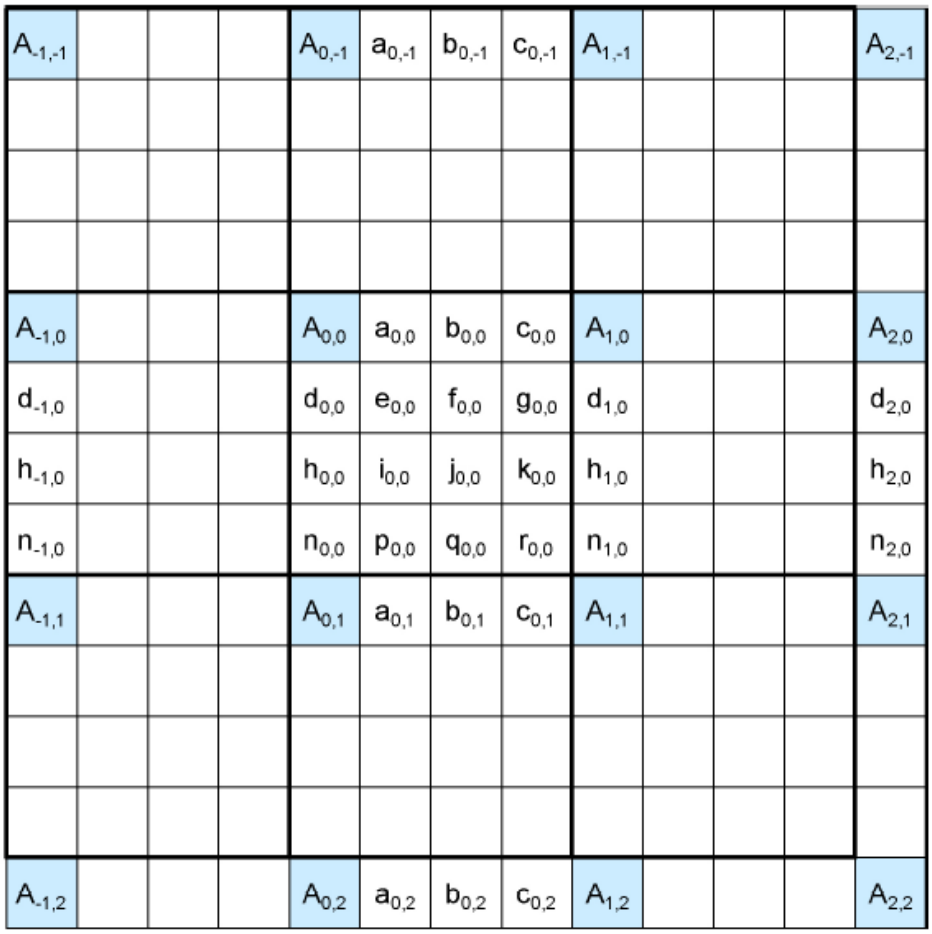

Figure 1.6: Fractional sub-sampled block with $1/4^{th}$ pixel accuracy

An example of integer and fractional sample luma interpolation is presented in figure 1.6. The $A_{i,j}$ position (upper-case letter) stands for an integer pixel, while fractional sub-pixels (lower-case letters) are obtained interpolating horizontally ($a_{0,j}, b_{0,j}, c_{0,j}$) and vertically ($d_{0,0}, h_{0,0}, n_{0,0}$) the integer neighboring pixels with equations presented in [3].

1.4 Hardware Architecture for Interpolation Filters

As explained in section 1.2.2 an hardware implementation for Interpolation filters is required in order to speed up the encoder/decoder execution time. The reference architecture that is the basis of this work of thesis can be retrieved open-source at [7].

1.4.1 2D DCT-IF Legacy Architecture Rescheduling

Since each luma and chroma prediction block interpolation is executed by means of two separable 1-D filters, first for the horizontal and then for the vertical direction, a buffer seems to be necessary between the two filtering operations: this will produce a very high throughput with the drawback of a big amount of required memory (as represented in figure 1.7).

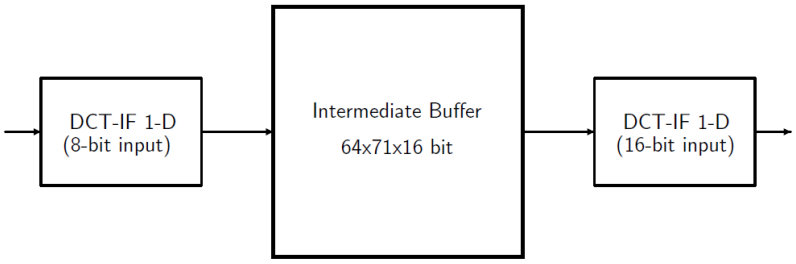

Figure 1.7: Parallel Interpolation filter architecture with intermediate buffer

The basic idea which is proposed in [2] is explained as following: it's not strictly necessary to wait for the entire prediction block to be partly sub-sampled in order to start a new filtering process. Thus, this can lead to a different scheduling: the first filter begins as soon as three data are ready as input and it sub-interpolates one pixel per cycles: the same mechanism is followed by the second stage filter that receives as input the pixels coming out from the first stage. In this way the required memory is reduced, because there's no need to store an entire prediction block in a buffer. If the buffer is placed before the 1^{st} stage filter, a throughput of 1-pixel per cycle is reached independently from the dimensions of the prediction block.

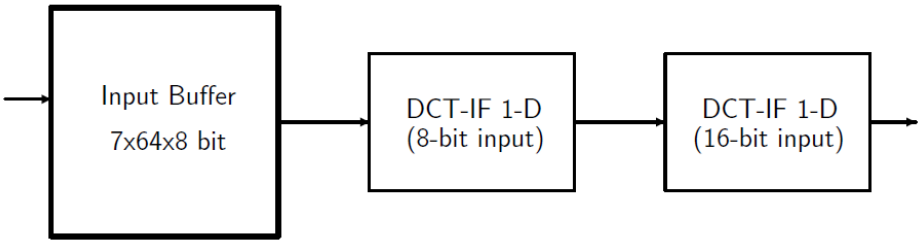

Figure 1.8: Parallel Interpolation filter with rescheduled architecture

In the proposed architecture, instead of proceeding with the original firstly horizontal and then vertical scheduling, the involved rescheduling provides columns as input of the first stage filter and rows for the second stage. In this way the same throughput is reached, with a further reduction in memory storage: in particular a memory reduction of 18.3x is gained with

respect to the original parallel architecture (figure 1.7) and 1.8x less memory is occupied than the original horizontal and then vertical HEVC scheduling. The new scheduling execution flow is presented in figure 1.9.

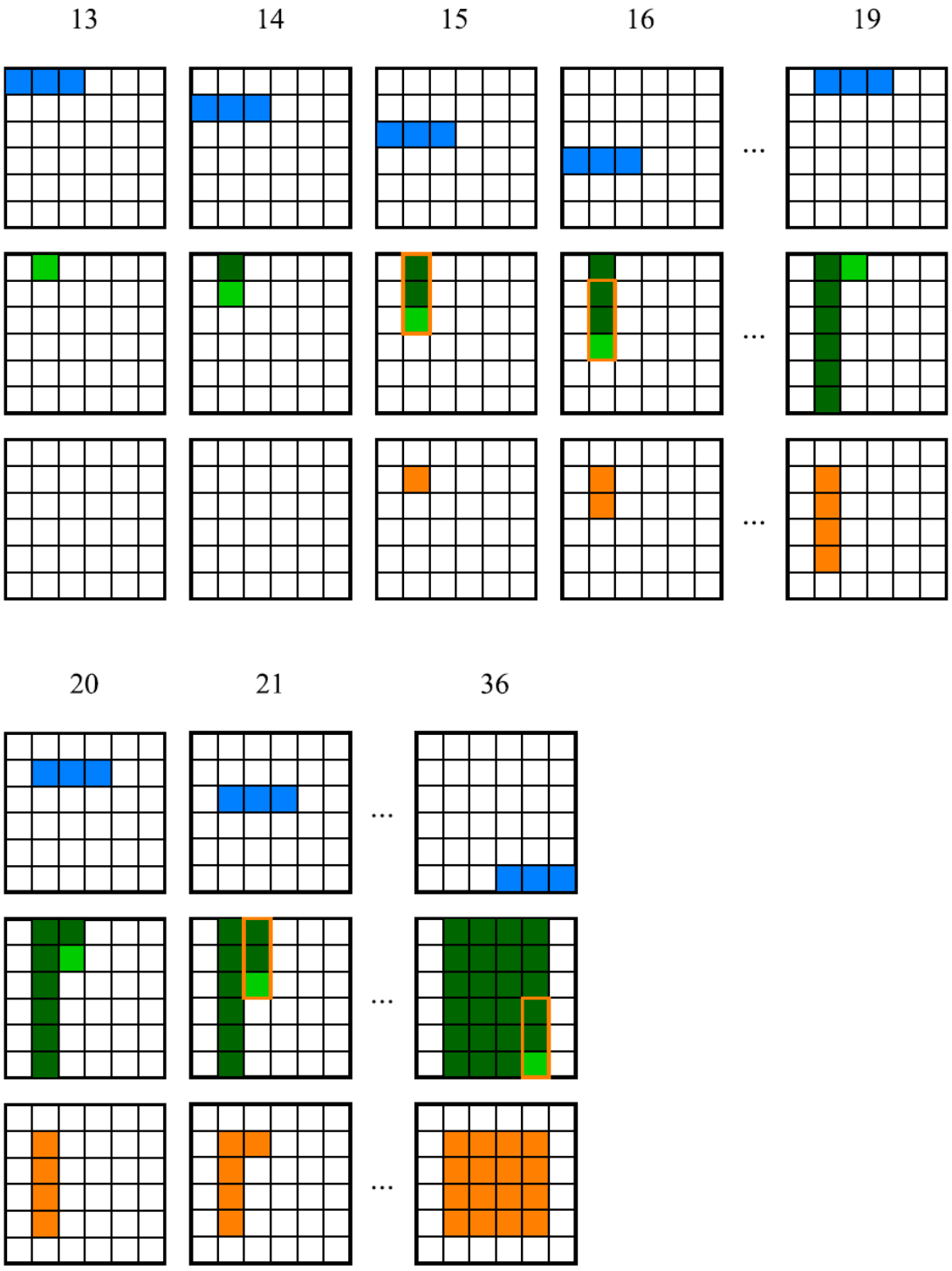

Figure 1.9: Filter vertical and horizontal rescheduling, figure courtesy [2]

1.4.2 A multiplier less solution for DCT-IF 1D Architecture

A Discrete Cosine Transform is based on a FIR filter standard architecture: this consists in the accumulation of different multiplications by fixed coefficients of a series of samples shifted in time domain:

$$y[n] = \sum_{i=0}^{N} B_i \cdot x_i[n-i] \tag{1.1}$$

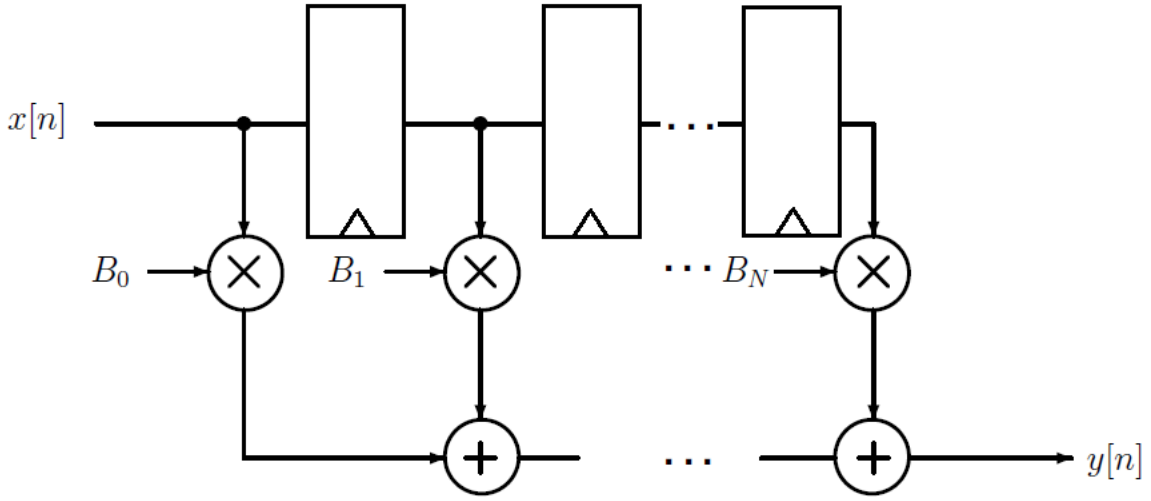

Figure 1.10: FIR Filter standard architecture

Since multiplications represent a higher computational cost with respect to sum, a multiplier-less solution is proposed similarly to [8], in order to shorten the critical path by increasing the throughput. The basic idea consists in replacing the starting architecture presented in 1.10 with an alternative that employs no more than additions and shift operations. Therefore, changing the internal architecture is responsible for a modification of the filter coefficients with sum and shifts, as presented in table 1.3.

shifts | coeff	**1**	**2**	**4**	**5**	**6**	**10**	**11**	**16**	**17**	**28**	**36**	**40**	**46**	**54**	**58**
x	+			+			+		+						
$x << 1$		+			+	+	+						-	-	+
$x << 2$			+	+	+					-	+				
$x << 3$						+	+					+		-	-
$x << 4$								+	+				+		
$x << 5$										+	+	+	+		
x « 6														+	+

Table 1.3: New coefficients table for sum and shifts

The new proposed sum and shifts DCT architectures are reproduced in figures 1.11 and 1.12, where the black numbers represent the first stage filter while the blue ones stand for the second stage filter.

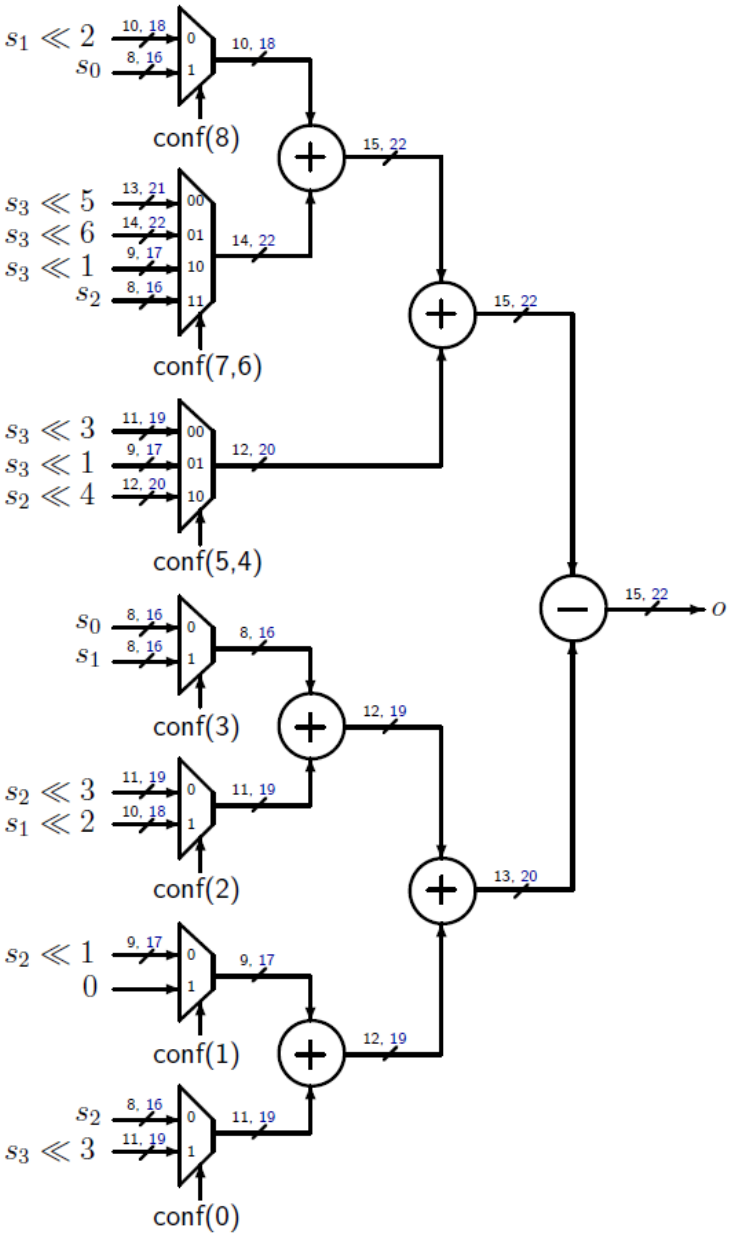

Figure 1.11: FIR Reconfigurable Luma Legacy Half Filter [2]

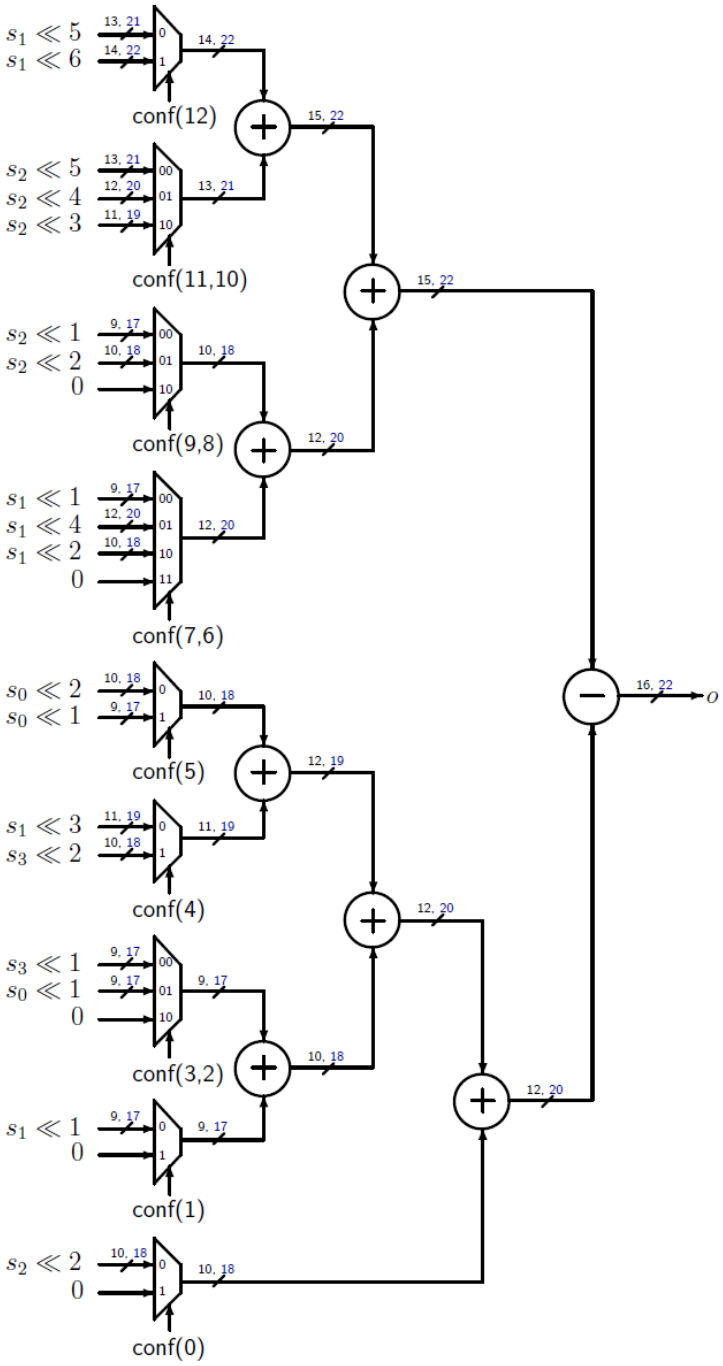

Figure 1.12: FIR Reconfigurable Chroma Legacy Filter [2]

1.4.3 Architecture Implementation

Datapath

The 2D interpolation filter datapath is composed by several elements, as figured in picture 1.13.

- **Address counter (CNT)**: is a programmable counter that has the purpose to point a SRB shift register. In order to start the filtering process, the lines to be used are filled by this counter;
- **Shift Register Bank (SRB)**: represents the input buffer. A shift register is placed at each row of the SRB. Once he receives as input the row to be provided to the routing unit from the counter, the pointed shift register shifts its content;
- **Routing Unit (RtU)**: used to redirect the output of the memory bank toward the correct inputs of the filter;
- **DCT-IF**: consists in the reconfigurable architecture presented in the previous subsection (figures 1.11 and 1.12);
- **Shift Register (SR)**: has the purpose to temporarily store data coming out from the first stage filter. A Serial Input Parallel Output Register is employed;
- **Rounding Unit**: applies a round to the half-up, if required, at the output of the second stage DCT-IF filter;
- **Clipping Unit**: necessary to deal with saturation arithmetic if a filter output on 8-bit unsigned is required.

FSM

The control unit is composed by a programmable counter (SCNT) that is shared between two main Finite State Machines (as represented in figure 1.14):

- **FSM1**: handles both 1D and 2D filtering operations since it controls the first stage filter(8 bits input). It's also in charge to provide the starting signal to FSM2 if a second interpolation is required. Otherwise it directly provides the output setting if a 1D interpolation is demanded by the user;
- **FSM2**: necessary to handle a 2D filtering operation, because it controls the second stage filter (16 bits input);
- **Shared Counter (SCNT)**: used both by FSM1, to count how many lines are filled by the Shift Register Bank, and by FSM2, to control how many partly interpolated filters are present in the 2^{nd} stage shift register. The Shared Counter can be used with muxes to handle its sharing, because lines count by FSM1 and FSM2 isn't simultaneously.

Further informations on the detailed implementation of the two FSMs can be retrieved in [2].

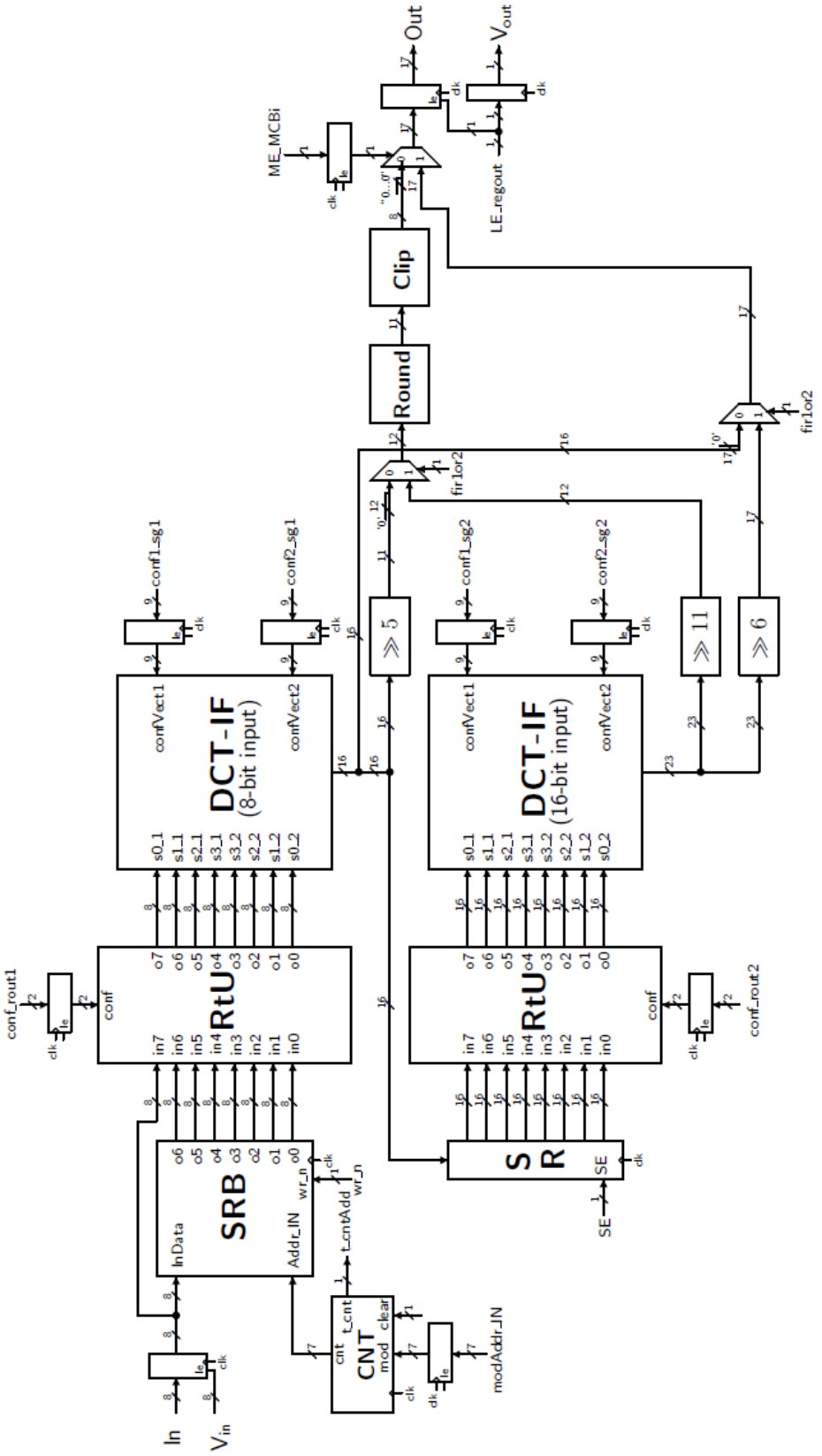

Figure 1.13: Datapath Luma Legacy [2]

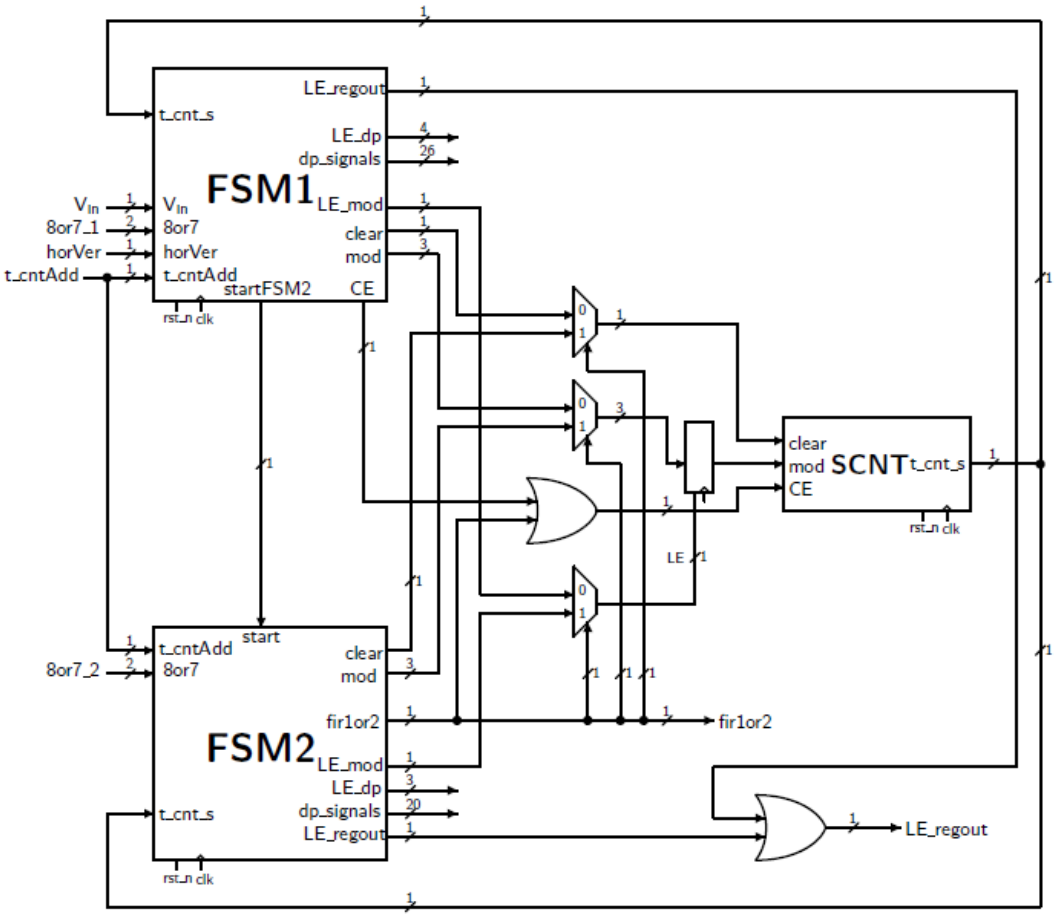

Figure 1.14: Control Unit Filter Legacy [2]

1.4.4 Processing Element

The Datapath and the Control Unit constitute the final Processing Element (PE) at the top level of the luma computation. Receiving as input a set of signals, it provides as output a signal V_{out} in correspondence of a valid output of the filter.

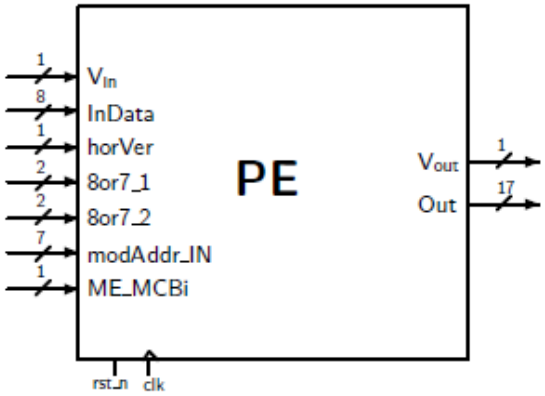

Figure 1.15: Luma top level Processing Element

Here a list of its input signals:

- V_{in}: signal that is sampled to start the filtering process;
- *InData*: is 8-bit unsigned pixel input;
- *horVer*: if it's 0 a 1D filtering is required, otherwise a 2D filtering is necessary;
- *8or7, 7or8*: sets the interpolation filter to be selected for the two filtering stages. If "00" a 8-tap half-pel filter is required, else if "10" a 7-tap $1/4^{th}$ quarter-pel filter, else if "01" a 7-tap $3/4^{th}$ quarter-pel filter, otherwise the FSM reaches the idle state;
- *ModAddr_IN*: determines the input modulus of the Adder Counter;
- *ME_MCBi*: if '0' the out is 8-bit unsigned, otherwise the full precision is required and a 16-bit output is provided.

Even if this final solution is able to provide higher throughput than the original software architecture, some improvements can be applied at different levels of the design. The purpose of this work of thesis is to find an appropriate internal architecture for the adders that are involved in the filtering operation, in order to further increase the throughput of the system.

Chapter 2

Chroma Legacy Architecture

Chroma Interpolation shows a similar process to the luma components. Since this signal is smoother with respect to the luma, four four-tap filters are required. In this chapter different solutions are applied to the adders that compose the second stage of filtering, which is the most time consuming part because a significant number of bits is involved.
After a first analysis of input data statistics, a simple solution that consists in cutting the LSBs of each adder and subtractor component of the second stage filter is applied and its results in terms of precision are analyzed at the output of the entire HEVC structure. Then parallel-prefix topologies are introduced to solve the time efficiency issue, whereas this first solution is too lossy in terms of precision of the system.

2.1 Data Analysis

A simulation of the entire behavior of the chroma legacy interpolation filters architecture is carried out in order to extrapolate input values for every adder inside the circuit. In this way, a histogram plotting the input data distributions is obtained per each sum and the mean and the standard deviation are computed in order to evaluate data trends from a quantitative point of view. These two figures of merits are figured out throughout the following formula, given an input array A of N samples:

$$\begin{aligned} \mu &= \frac{1}{N}\sum_{i=1}^{N} A_i \\ \sigma &= \sqrt{\frac{1}{N-1}\sum_{i=1}^{N} |A_i - \mu|^2} \end{aligned} \tag{2.1}$$

In particular, standard deviation is essential in order to get a handle on whether the data are close to the average or if they are spread out over a wide range. Since the initial idea involves the application of Variable Latency Adders to improve the throughput of the system, there is a strict relation between VLAs' performances and the assumptions concerning input signal statistics: in particular, the error rate increases for a uniform distribution with respect to a half uniform and half Gaussian statistic with a low enough standard deviation.
Hence, a sequence of different input files have been executed one after the other in a single simulation, in order to estimate the input statistics for different combinations of requested sub-pixels, and to reproduce the software behavior of the architecture. Results in terms of

evaluated figures of merits and a slice of the obtained data statistics are presented in table 2.1 and figure 2.1.

	Mean μ	Std_Dev σ
FS2_op1_A	361685.36	117794.10
FS2_op1_B	148893.35	75044.59
FS2_op2_A	19277.23	6933.43
FS2_op2_B	58379.34	47596.22
FS2_op3_A	24875.84	6680.12
FS2_op3_B	45145.87	14816.59
FS2_op4_A	14387.09	1140.54
FS2_op4_B	14382.55	1074.29
FS2_op5_A	510536.00	59661.13
FS2_op5_B	69800.18	39375.84
FS2_op6_A	70015.84	12993.40
FS2_op6_B	24114.41	6956.69
FS2_op7_A	90634.36	19416.66
FS2_op7_B	28765.10	2148.81
FS2_op8_A	560143.67	43205.23
FS2_op8_B	98954.85	21945.68

Table 2.1: Mean and Standard Deviation for Chroma Legacy Input Data Adders second stage filter

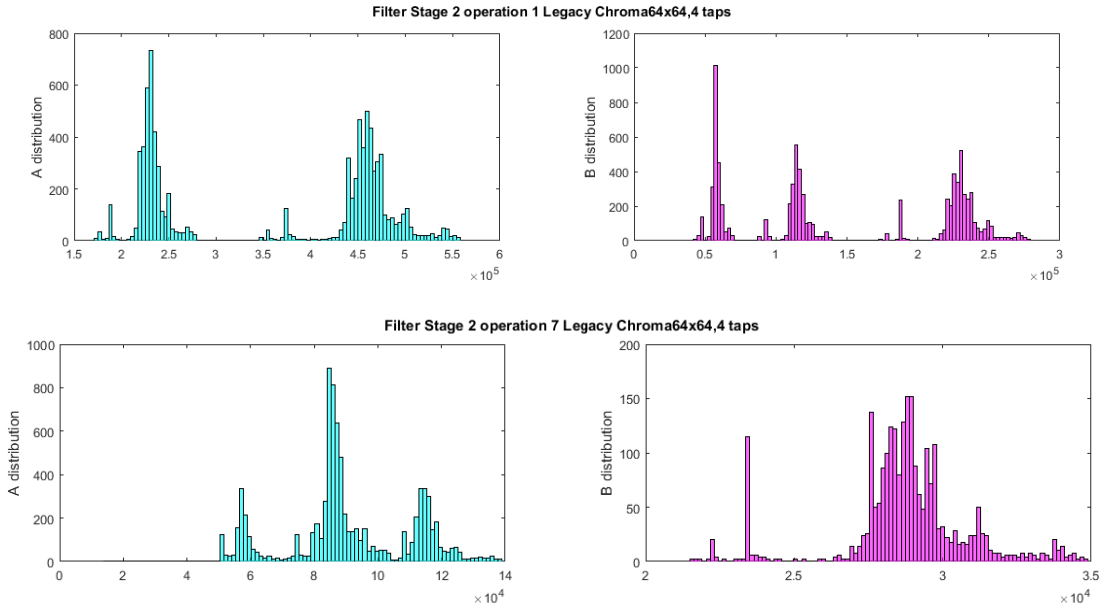

Figure 2.1: Chroma Legacy Partial Input Data Statistics

2.2 Cut Architecture

From the analysis of the statistics of the input data of each adder of the second stage filter, it can be noticed that almost all of them are within the range $[10^4 : 10^6]$: this corresponds to additions and/or subtractions that could be theoretically computed for values down to the 13^{th} bit as upper limit.

2.2.1 Filter Design

Several versions have been developed, starting from the original addends of the operations and cutting from each input of every single adder a fixed number of bits. This fixed amount of bits is obtained by removing the LSBs for each sum to achieve a lower (which implies in principle faster) parallelism, as depicted in figure 2.2. This straight-forward operation is repeated for an increased number of bits: the first version consists in cutting the 10 Least Significant Bits from each adder input, up to a last version that involves a 13 bits truncation. It is expected that cutting a higher number of bits will lead to a less and less precise result, so a measure of the signal degradation must be carried out.

2.2.2 Interpolation Filter Output

For each truncation, a simulation of the expected filter output is implemented: some significant samples among filter outputs are depicted in figures 2.3 and 2.4.

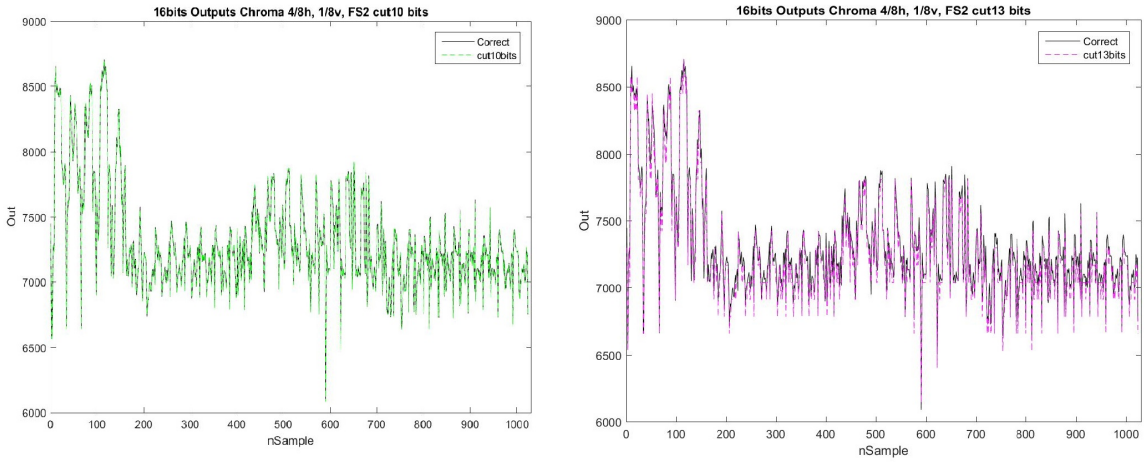

Figure 2.3: Chroma Legacy Filter 4/8h 1/8v Output for 10 and 13 bit cuts

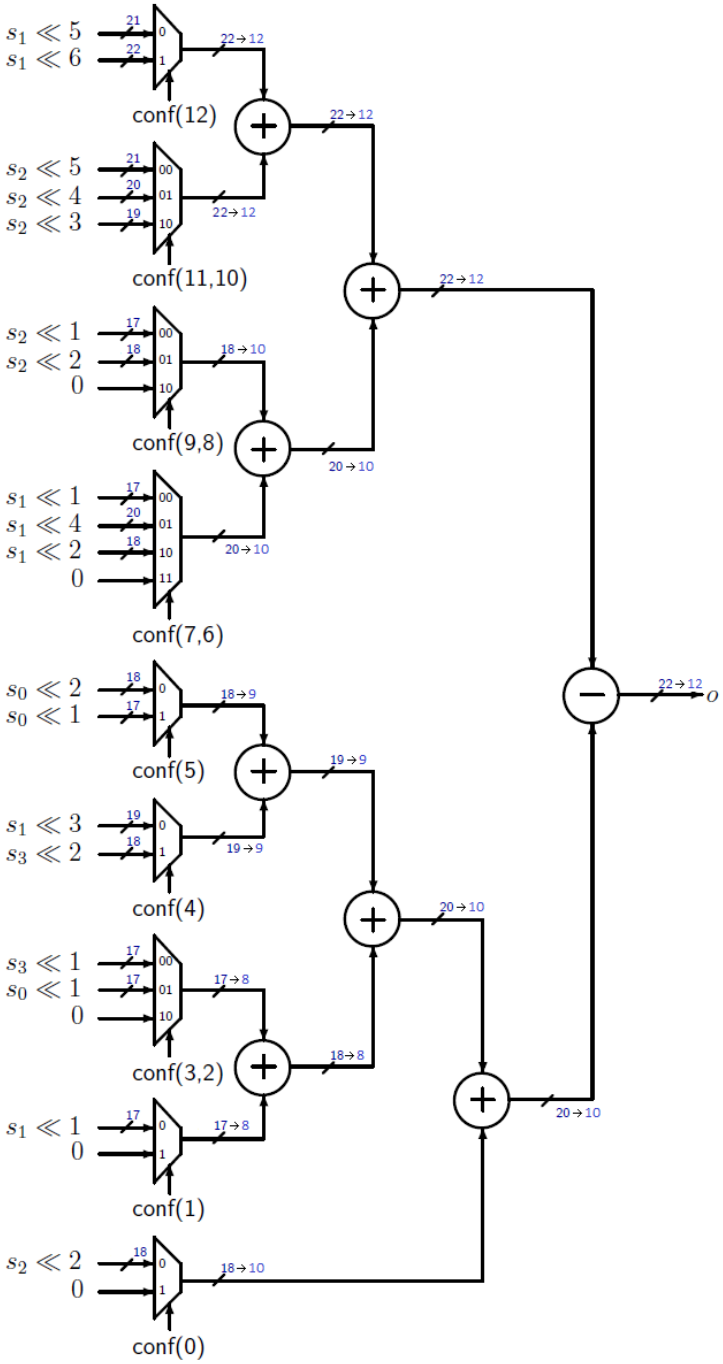

Figure 2.2: FIR Reconfigurable Cut Chroma Legacy Filter

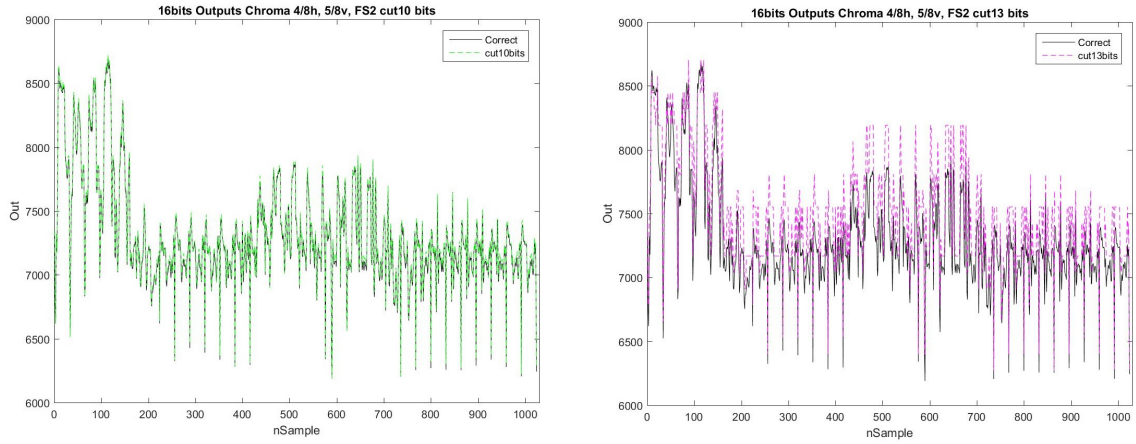

Figure 2.4: Chroma Legacy Filter 4/8h 5/8v Output for 10 and 13 bit cuts

As far as simulations are concerned, the Root Mean Square Deviation has been computed in order to evaluate how much the signal out of the filter is degraded due to the cut operation:

$$RMSD = \sqrt{\frac{\sum_{i=1}^{n}(x_{i,correct} - x_{i,cut})^2}{n}} \tag{2.2}$$

with n the number of samples.

Simulations are performed for two different conditions and the obtained results for each simulation are reported in the following tables:

- RMSD for 8 bits out: the output is provided as 8-bit unsigned, as the input format, passing throughout the rounding and clipping unit, as it's required by both Motion Compensation and Motion Estimation

	4/8v,4/8h	*4/8v,1/8h*	*4/8v,2/8h*	*4/8v,3/8h*	*4/8v,5/8h*	*4/8v,6/8h*	*4/8v,7/8h*
cut10	0.40384	0.36577	0.62029	0.64424	0.6607	0.63199	0.3737
cut11	0.56596	0.50967	0.9956	1.0602	1.0486	1.0078	0.50292
cut12	1.2543	0.75584	1.2824	1.4717	1.4856	1.3188	0.73686
cut13	2.1653	1.4201	3.256	3.6966	3.6689	3.3254	1.356

Table 2.2: Chroma Legacy Comparison RMSD for different LSBs cuts, 8 bits out

- RMSD for 16 bits out: the output is kept to higher precision, without being affected by any clipping/rounding operations, as it is required by the Motion Compensation bi-prediction case

	4/8v,4/8h	*4/8v,1/8h*	*4/8v,2/8h*	*4/8v,3/8h*	*4/8v,5/8h*	*4/8v,6/8h*	*4/8v,7/8h*
cut10	17.158	12.287	22.006	24.923	25.018	22.064	12.553
cut11	36.562	26.428	45.722	50.979	51.195	45.856	26.563
cut12	77.423	41.924	80.728	92.152	92.979	82.433	41.769
cut13	136.94	86.815	208.98	235.76	233.39	212.21	83.359

Table 2.3: Chroma Legacy Comparison RMSD for different LSBs cuts, 16 bits out

2.2.3 HEVC Output

An analysis of the coding efficiency of the entire HEVC system is executed in order to quantify the effects of the cutting operations on the final outcome of the system with respect to the case with absolute precision. A slight modification is performed on the software reference code [6] in order to let it perform the same cut operations as the proposed hardware architecture. The quality and the bit-rate data are exploited to plot the rate-distortion curve of an encoder: the Peak Signal to Noise Ratio (PSNR) is calculated as metric to quantify the quality of the encoder. Per each frame the combined $PSNR_{YUV}$ is computed as the weighted sum of a luma ($PSNR_Y$) and two chroma ($PSNR_U$, $PSNR_V$) signals [9]:

$$PSNR_{YUV} = \frac{6PSNR_Y + PSNR_U + PSNR_V}{8} \tag{2.3}$$

where each component (e.g. $PSNR_Y$) is evaluated as:

$$PSNR_i = 10\log_{10}\left(\frac{(2^B-1)^2}{MSE}\right), \qquad i = Y,U,V \tag{2.4}$$

The MSE expresses the mean square error, that is related to the previous results which are obtained as output of the filter simulations. For each simulation, the total PSNR is calculated as the average of the PSNR of each frame.
The obtained curve interpolates four rate-distortion points, which are collected for different QP values (22, 27, 32, 37), with a shape-preserving piecewise cubic interpolation. As it can be shown from the obtained results, Motion Estimation is strongly influenced by the quantization parameter: if a small QP is chosen (-q 22) small variations are more likely captured and a higher bitrate is required. As a consequence, the higher is the bitrate, the higher will be the complexity of the filter.
The HEVC reference encoder supports three distinct kinds of configurations, each of them relies on a different prediction structure:

- **All Intra**: all frames are encoded as I-type. There is not the adoption of any temporal prediction and the QP value doesn't change between consecutive frames;
- **Low Delay**: frames are encoded in order. The first frame is encoded using I-slices, while all the successive frames are of B or P type. It emulates videoconferencing environment;
- **Random Access**: frames are encoded through a picture reordering with random-access of the figure. Frames are only of I or B type. It mirrors the broadcasting environment.

Simulations are carried out for Low Delay and Random Access configurations for the reference file *BasketBallDrive.yuv* as depicted in figures 2.5 and 2.6.

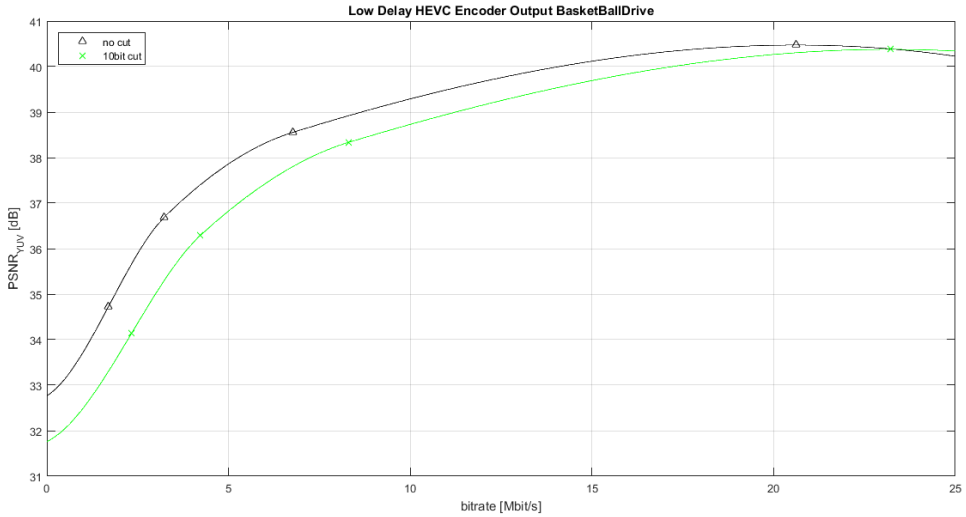

Figure 2.5: PSNR degradation with Chroma cut computing (BasketballDrive[17], 1920x1080, 50 Hz, Low Delay)

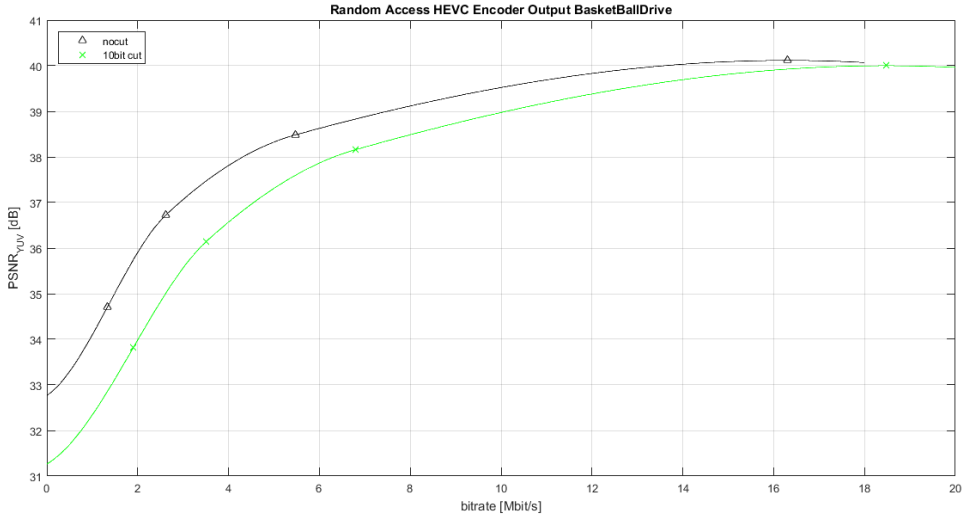

Figure 2.6: PSNR degradation with Chroma cut computing (BasketballDrive[17], 1920x1080, 50 Hz, Random Access)

As it can be observed from pictures above, the cut approximation is a too rough computation inside the interpolation process, that causes a high degradation of the output filter performances. Thus, this solution is discarded because a more precise implementation is required.

2.3 Parallel & Prefix Adders

The general starting structure of an adder computes the sum of two n-bits addends A=$a_{n-1}a_{n-2}...a_0$ and B=$b_{n-1}b_{n-2}...b_0$ through the following expressions:

$$\begin{aligned} s_i &= a_i \oplus b_i \oplus c_{i-1} \\ c_i &= a_i b_i + a_i c_{i-1} + b_i c_{i-1} \end{aligned} \tag{2.5}$$

This implements a general structure like a Ripple-Carry Adder (RCA): the main problem with this configuration concerns timing, since the carry computation represents the bottleneck in the critical path evaluation. Parallel Prefix Adders (PPAs) have been introduced in order to speed-up performances of this operation. Indeed they achieve different trade-offs in terms of speed, complexity and fan-out.

2.3.1 General Structure

In parallel-prefix addition the sum operation is split in three main steps:

1. *Pre-Processing*: generate and propagate bits are derived from addends bits:

$$g_i = a_i b_i \qquad p_i = a_i \oplus b_i \tag{2.6}$$

2. *Prefix-Processing*: solving the parallel-prefix problem all the G,P couples are obtained from 0 to all the possible positions:

$$(g_{[i:k]}, p_{[i:k]}) = (g_{[i:j]}, p_{[i:j]}) \& (g_{[l:k]}, p_{[l:k]}) = (g_{[i:j]} + g_{[l:k]} p_{[i:j]}, p_{[i:j]} p_{[l:k]}) \qquad i \geq l \geq j \geq k \tag{2.7}$$

3. *Post-Processing*: carry is computed from G,P and exploited to determine the output sum bits:

$$c_i = G_{[i:0]} + P_{[i:0]} c_0 \qquad s_i = p_i \oplus c_{i-1} \tag{2.8}$$

Among different blocks, the prefix-processing represents the most complex part: it consists into a network where the operator & is combined in such a way that all the generate and propagate terms extended to blocks of contiguous bits. What characterizes parallel prefix adders is the fact that all the carry bits are computed in parallel, provided that the associative property is satisfied.
Different kinds of PPAs [10] are distinguished depending on the way the tree of generating and propagating bits is organized:

- *Brent-Kung* : is composed by the lowest number of generate and propagate units, that results into a low area. However it is characterized by a maximum logic depth that implies on higher delay;
- *Kogge-Stone* : is built of a very high number of propagate and generate blocks but a minimum fan-out. This results in large occupied area but low delay;
- *Han-Carlson* : aims to reach a trade-off between the characteristics of the Brent-Kung and Kogge-Stone topologies. Indeed it is intended to exploit both the lower area of the first one and the higher speed of the latter;

- *Ladner-Fischer* : wants to reach a lower depth of critical path, at the cost of a higher fan-out;
- *Sklansky* : shows similar characteristics to the Ladner-Fischer, what changes is just the inner configuration.

Table 2.4 summarizes the trend of area and time for different adders:

	Area	**Delay**
Brent-Kung	$2n - 2 - \log_2(n)$	$2\log_2(n) - 2$
Kogge-Stone	$n\log_2(n) - (n-1)$	$\log_2(n)$
Han-Carlson	$\frac{n}{2}\log_2(n)$	$\log_2(n) + 1$
Ladner-Fischer	$2n - 2 - \log_2(n)$	$2\log_2(n) - 1$
Sklansky	$\frac{n}{2}\log_2(n)$	$\log_2(n)$

Table 2.4: Performances and Area of Parallel&Prefix Adders

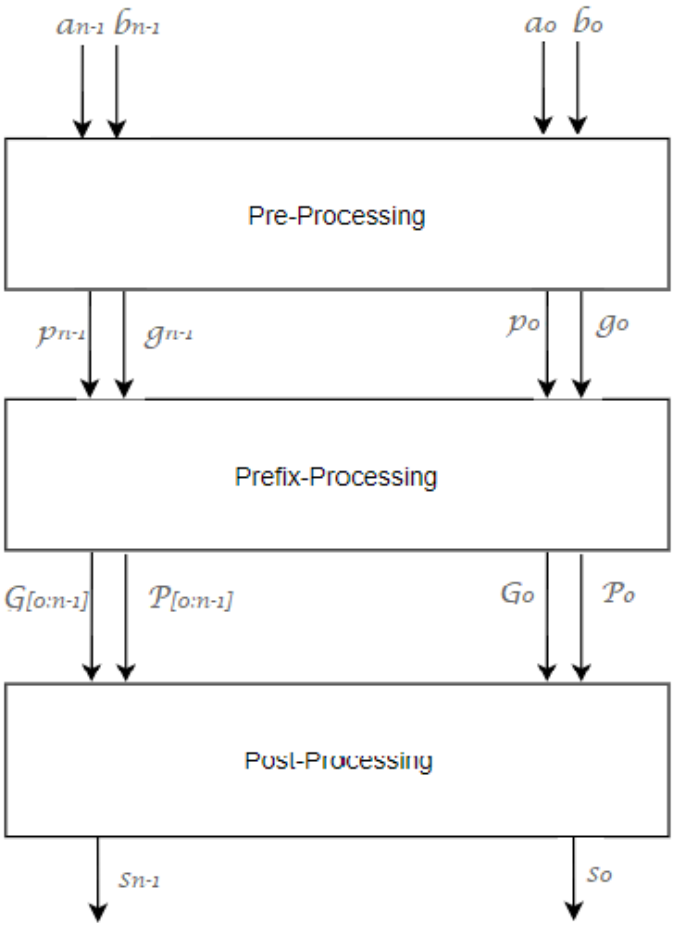

Figure 2.7: Parallel & Prefix Adder Block Diagram

2.3.2 Han-Carlson Adder

The Han-Carlson adder represents a good trade-off between complexity, fanout and performances. It's mainly divided in two blocks: while the outer rows are Brent-Kung graphs (blue circles), the inner ones are Kogge-Stone type (white circles). In this way it is able to reach the same speed performance as the Kogge-Stone by dissipating a lower power and occupying a lower area.
The general scheme is presented for a number of bits equal to 22 as represented in figure 2.8 (that is the maximum number of bits to be applied in the proposed architecture), which can be easily applied to a lower number of bits by pruning MSBs columns.

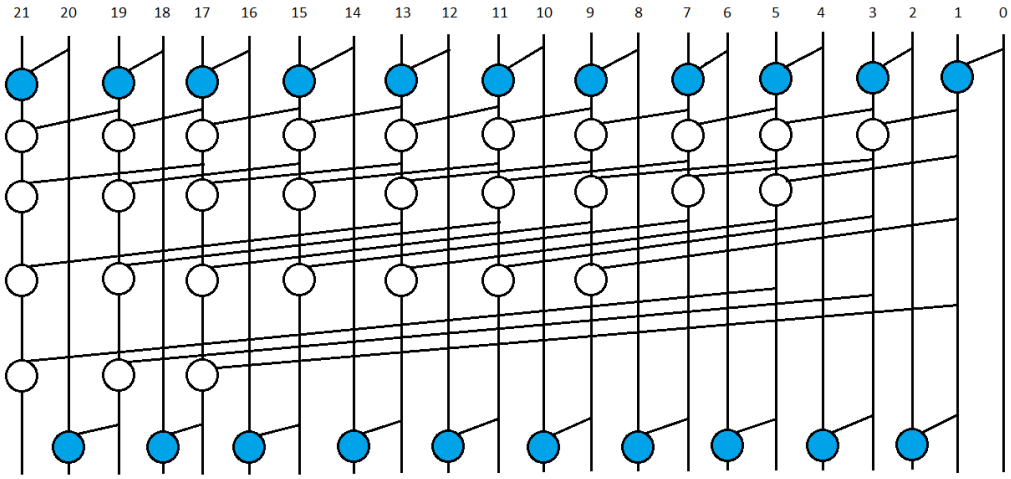

Figure 2.8: Han-Carlson parallel-prefix block

Approximate Han-Carlson Architecture

Two approximate versions of the starting Han-Carlson parallel and prefix architecture are proposed: the first one is obtained by deleting the very last row of Kogge-Stone adder, while the second one by removing the two last rows of the Kogge-Stone. The two alternative structures are presented in figures 2.9 [11] and 2.10.

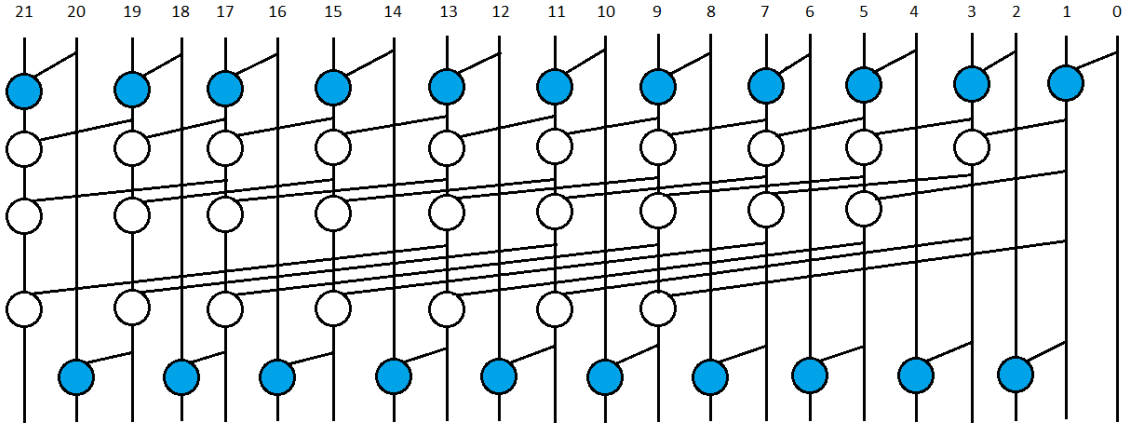

Figure 2.9: Approximate Han-Carlson version 1

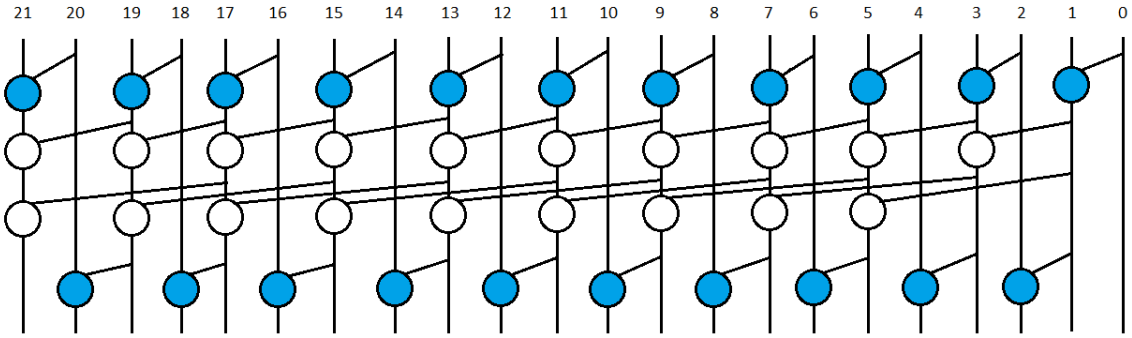

Figure 2.10: Approximate Han-Carlson version 2

2.3.3 Ladner-Fischer Adder

This kind of adder has been introduced in order to shorten the critical path of the tree of prefix operators together with the Sklansky adder. It presents a reduced critical path with the drawback of a higher fan-out, which is a crucial issue in timing, since the delay is directly proportional to the fan-out of a gate.
A slightly different version of the original adder is proposed [12], where the first two levels and the last one are Brent-Kung type (blue circles), while the inner rows are Ladner-Fischer topology (white circles), as depicted in figure 2.11.

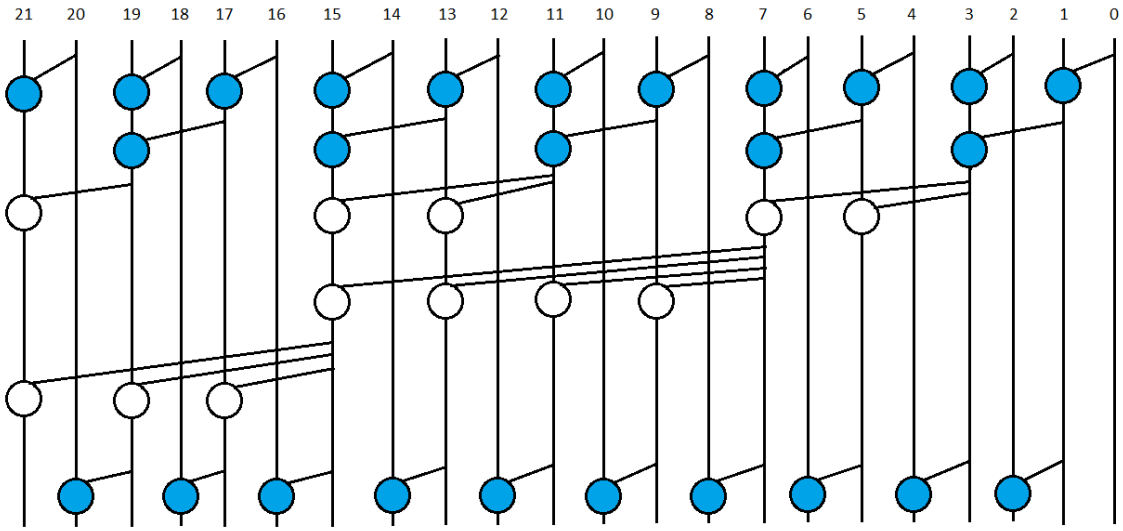

Figure 2.11: Ladner-Fischer parallel-prefix block

Approximate Ladner-Fischer Architecture

The approximate version of the architecture is retrieved by deleting the last two rows of the Ladner-Fischer part and by modifying the remaining third row of the parallel-prefix block in picture 2.12.

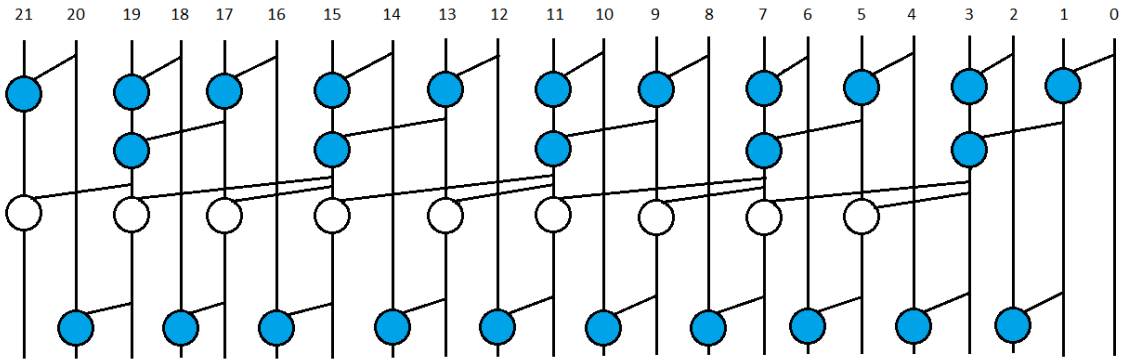

Figure 2.12: Approximate Ladner-Fischer parallel-prefix block

Referring to the results of paper [12], whose synthesis outcomes have been obtained employing the same technology as the one that is adopted in the proposed work of thesis, it is expected that the Ladner-Fischer outperforms in performances the Han-Carlson architecture for adders whose maximum number of bits is 32.

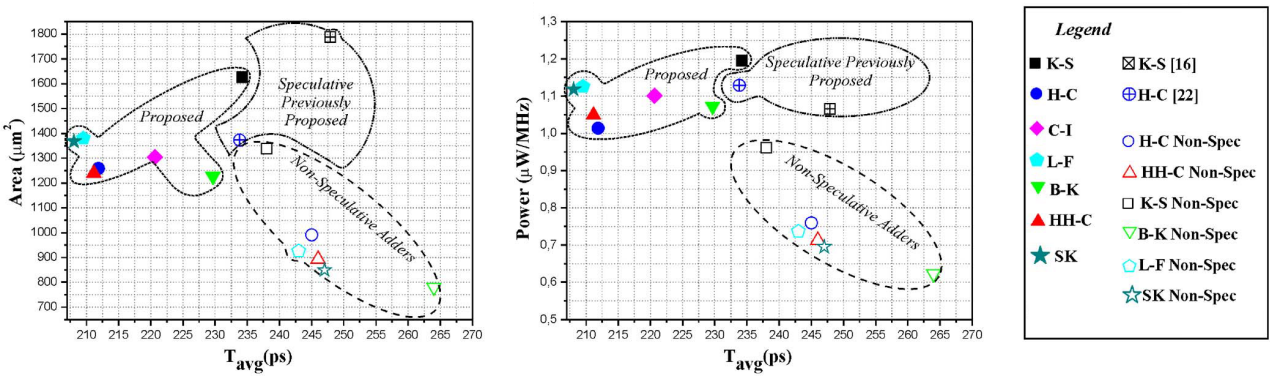

Figure 2.13: Approximate Ladner-Fischer and Han-Carlson comparison for 32-bits output

However the required number of bits for the performed operations is variable and lower or equal to 22 bits, so simulations are required to return a good estimation of the proposed architecture.

2.3.4 Interpolation Filters Output

The proposed adder architecture is applied to all the sums that characterize the second stage filter: this one is employed for the 2D interpolation and it is characterized by a higher number of bits with respect to the first stage.
Hence, all the different input files have been executed one after the other in a single simulation, in order to make an assessment of the RMSD and of the number of errors that are present on a total amount of 7168 samples and to compare the two different versions of approximate adders.

	RMSD	# errors	% errors
HC_Approximate v1	0	0	0%
HC_Approximate v2	296.78	104	1.45%
LF_Approximate	509.14	264	3.68%

Table 2.5: Chroma Legacy Comparison RMSD, 16 bits out

As expected the Ladner-Fischer approximate architecture shows a higher error bit-rate with respect to the Han-Carlson topology, a drawback which is necessary in order to obtain better results in terms of performances and area.
Since the first approximate version of Han-Carlson adder, where just one row is pruned, shows a null Root Mean Square Deviation, the version with two cut levels will be the one to be synthesized in the following steps.

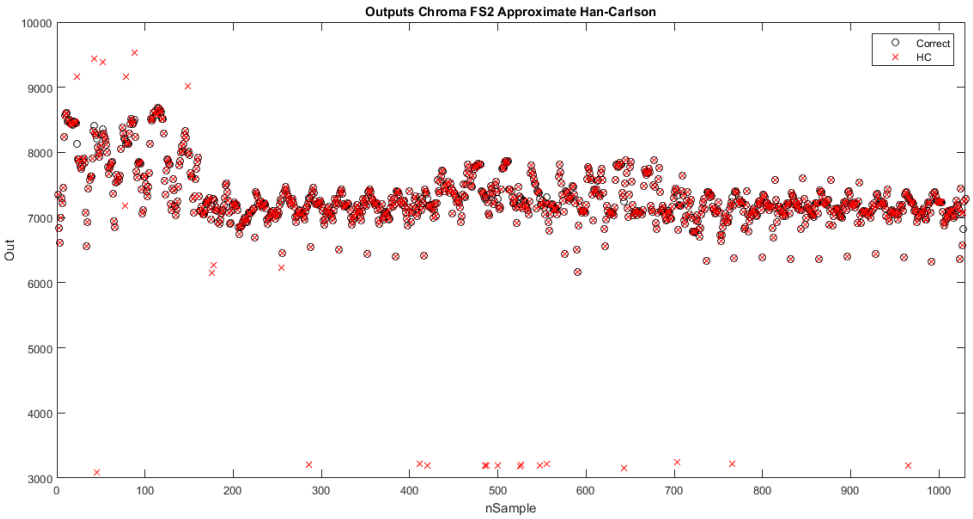

Figure 2.14: Chroma Legacy Filter Output, Han-Carlson Approximate Adder topology

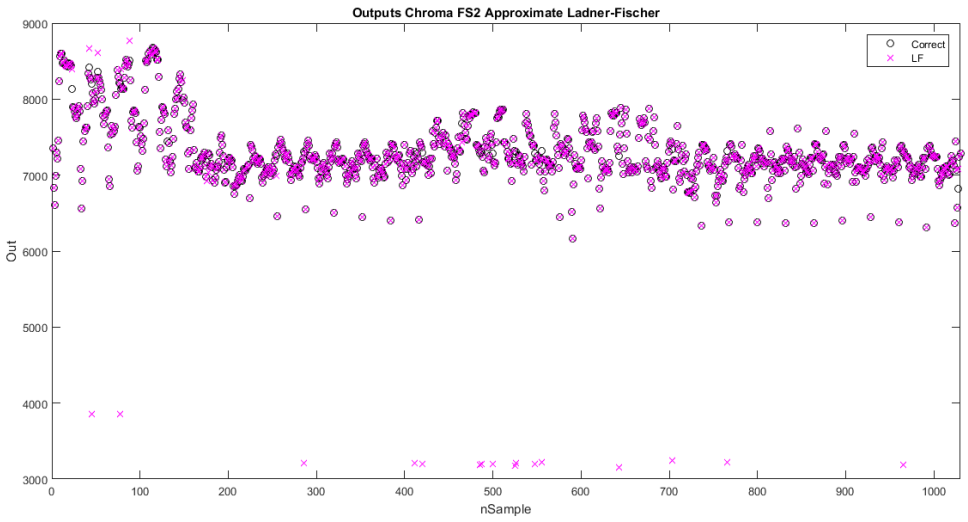

Figure 2.15: Chroma Legacy Filter Output, Ladner-Fischer Approximate Adder topology

2.3.5 Design Synthesis Results

Different versions of the implemented adders are inserted in the complete interpolation filter architecture which is synthesized in UMC 65 nm library at 1.2 V.
Some time constraints like input and output delay, clock uncertainty, output load from the chosen technology library, are set to provide a more realistic evaluation of power, area and timing for the proposed design. The clock gating technique is applied in order to improve performances and reduce the occupied area with a lower power consumption with respect to a non-gated solution.
Simulations have been performed for a 32x32 pixel block concerning chroma legacy architecture. Dynamic power dissipation is evaluated on the produced net-list after synthesis, by extracting the nodes activities from back-annotated simulation of *Mentor Modelsim* and computing power consumption with *Synopsys Design Compiler*.
The obtained synthesis results are reported in table 2.6. Precisely the f_{max} parameter is representative of the maximum operating frequency that allows to achieve a zero slack with the Design Compiler tool, while area and power are evaluated at the lowest between the operating frequencies of Luma Legacy and Chroma Legacy architecture, in order to provide accurate comparisons among different implementations.

	f_{max} [MHz]	Area [μm^2] [1]	Power [mW] [1]
PE correct	680.27	14822.64	2.966
PE HC correct	689.66	15457.32	3.040
PE HC approx	684.93	16388.28	3.494
PE LF correct	694.44	15366.60	3.013
PE LF approx	689.66	15804.36	3.176

Table 2.6: Chroma Legacy Filter Synthesis results with clock gating

It can be noticed that the Ladner-Fischer correct architecture shows the best improvements in performances, at the cost of a negligible area overhead and power dissipation, therefore it will be the one adopted for computation. The area increment with approximate architectures can be justified as a lower freedom for the synthesizer to optimize the design, given a lower number of available blocks; however it's not easy to understand the internal algorithms that *Synopsys DC* applies. As a consequence it is not necessary to apply a speculative approach to a system that shows better synthesis results with an exact architecture. Indeed the Ladner-Fischer correct topology shows the most suitable outcomes with an improvement in performances of 2.08% with a 3.66% area overhead and an increased 1.58% power consumption.

	Δf_{max} [%]	ΔA [%]	ΔP [%]
PE HC	+1.38	+4.28	+2.49
PE HC App	+0.69	+10.56	+17.80
PE LF	+2.08	+3.67	+1.58
PE LF App	+1.38	+6.62	+7.08

Table 2.7: Chroma Legacy relative percentage comparisons with the original design

[1] Area and Power are evaluated at t_{ck}=1.70 ns, that is the worst case for the entire interpolation filters architecture, i.e. luma legacy structure

Chapter 3

Luma Legacy Architecture

Differently from chroma design, in luma architecture each DCT-IF stage is composed of a mirrored structure: the same tree of adder is repeated two times to reply the behavior of a 8/7 order FIR filter, therefore a much higher complexity is considered with respect to the previous architecture. After a data analysis that highlights uniform statistics as input of each adder, the same strategy is adopted with Parallel & Prefix Adders in both exact and approximate structure as for the chroma case. Then multi-operand carry save adders are used to improve the system performances. As alternative approach, adaptive approximate computing with Generic Accuracy Configurable adders is applied.

3.1 Data Analysis

A first data analysis is executed as for the chroma legacy architecture, in order to understand which are the data statistics that characterize different adders in the Datapath.
As before, the attention is focused on the second stage filter, because a higher number of bits, so a more significant parallelism is concerned in the interpolation process. The same figures of merits, mean and standard deviation, whose expressions are reported in section 2.1, are evaluated in the current analysis.

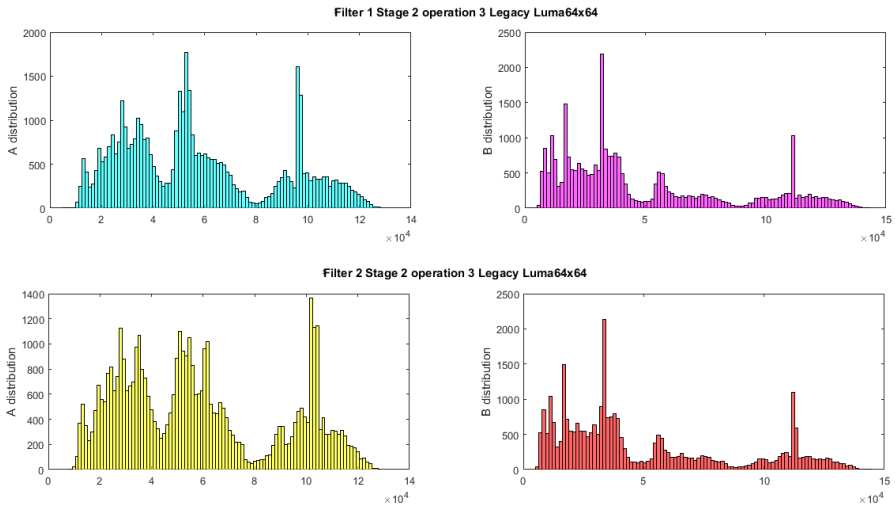

Figure 3.1: Luma Legacy Partial Input Data Statistics

	Mean μ	Std_Dev σ
F1S2_op1_A	22594.80	15385.50
F1S2_op1_B	241323.54	236912.59
F1S2_op2_A	263907.37	249696.62
F1S2_op2_B	66236.01	55632.48
F1S2_op3_A	57834.47	29693.66
F1S2_op3_B	48348.82	35814.56
F1S2_op4_A	7565.25	3354.41
F1S2_op4_B	50271.50	27152.30
F1S2_op5_A	15084.20	6695.82
F1S2_op5_B	33264.63	32462.33
F1S2_op6_A	330140.37	226679.46
F1S2_op6_B	89959.78	60880.38
F2S2_op1_A	23360.10	16021.03
F2S2_op1_B	249629.39	240984.54
F2S2_op2_A	272988.43	254350.90
F2S2_op2_B	64973.82	55198.69
F2S2_op3_A	58843.28	30487.74
F2S2_op3_B	49511.69	36452.86
F2S2_op4_A	7672.37	3475.38
F2S2_op4_B	51171.26	27794.15
F2S2_op5_A	15303.70	6808.13
F2S2_op5_B	34207.99	32954.44
F2S2_op6_A	337962.25	231828.03
F2S2_op6_B	92281.41	62508.84

Table 3.1: Mean and Standard Deviation for Luma Legacy Input Data second stage filter

Picture 3.1 and the high dispersion of standard deviation in table 3.1 underline that here is the case of a uniform distribution. This conveys that an approximate approach is expected be more efficient than speculation, since a larger amount of errors will characterize the architecture, as explained in paper [12].

3.2 Parallel & Prefix Architecture

A first idea is to apply the same approach as in the chroma legacy architecture, by replacing all the adders that compose the second stage filters (because the structure is mirrored, as already mentioned) with Parallel & Prefix architectures. Thus, both precise and approximate Han-Carlson and Ladner-Fischer topologies are applied to the DCT-IF datapath in order to improve the different characteristics of the interpolation filters. Then their performances are evaluated both in terms of precision and concerning area, speed and power consumption.

3.3 Carry Save Adder Architecture

The CSA is a topology of adder whose key idea consists in the employment of the full adder as a compressor. Indeed, given a full adder, all the inputs have the same weight (a_i, b_i, c_i): thus, for 3 different input operands of N bits, the output s_i per each bit has a weight equal to 2^i, while the output carry is 2^{i+1} and it will be propagated to an other full adder. In this way every stage is able to reduce the number from three to two operands.
If a multi-operand adder is concerned, a tree of compressor can be exploited with two different notations: either a Dadda tree or a Wallace tree can be handled to reach this goal.

3.3.1 Filter Design with CSA

Concerning the proposed architecture a Dadda tree is applied to the DCT-IF structure. This structure identifies the number of operands that can be added at every stage of the tree: starting from the bottom of the structure, with d_1=2, that is the number of wanted rows at the end of the tree, the number of operands that are required in the previous level follows equation 3.1 in order to derive the total number of levels in the tree:

$$d_{j+1} = \left\lfloor \frac{3}{2} d_j \right\rfloor \tag{3.1}$$

The Dadda approach is used to add half and full adders only if it is required: this will limit the hardware complexity and the delay of the structure with respect to the Wallace architecture. This last follows a ASAP approach, by allocating compressors each time tree lines of products are found.
The filter design for the Luma Legacy architecture is depicted in figure 3.5: in this case a multi-operand 3 to 1 tree of compressor is adopted for the upper inputs, while a 4 to 1 Dadda tree is handled for the remaining addends.
In a more detailed way, carry select adders are employed until the last level, while the last row is a generic two operands adder. The two structures are presented both with the schematic and with the dot notation in figures: 3.2, 3.3 and 3.4.

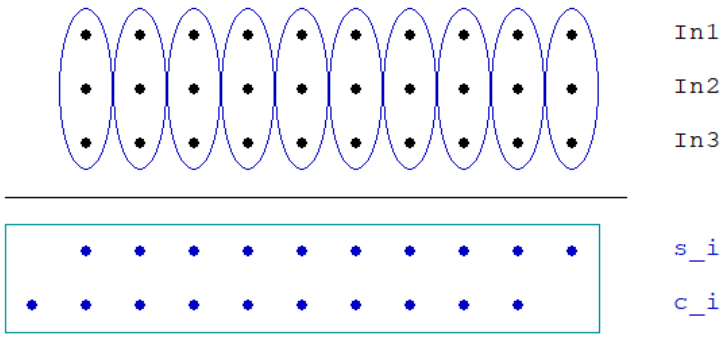

Figure 3.2: CSA tree 3 operands dot notation

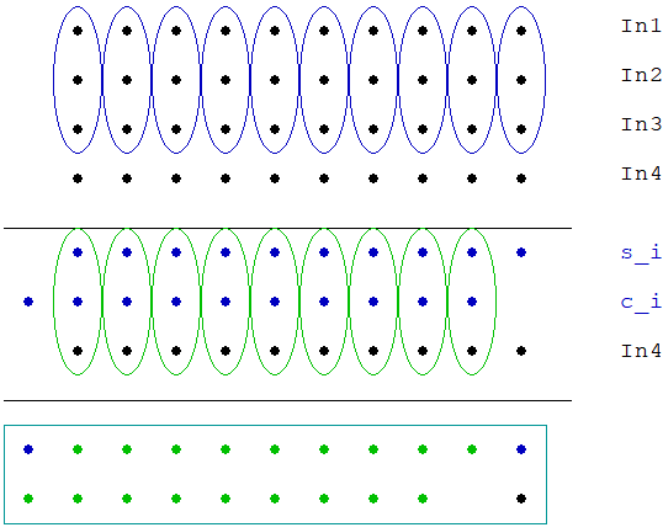

Figure 3.3: CSA tree 4 operands dot notation

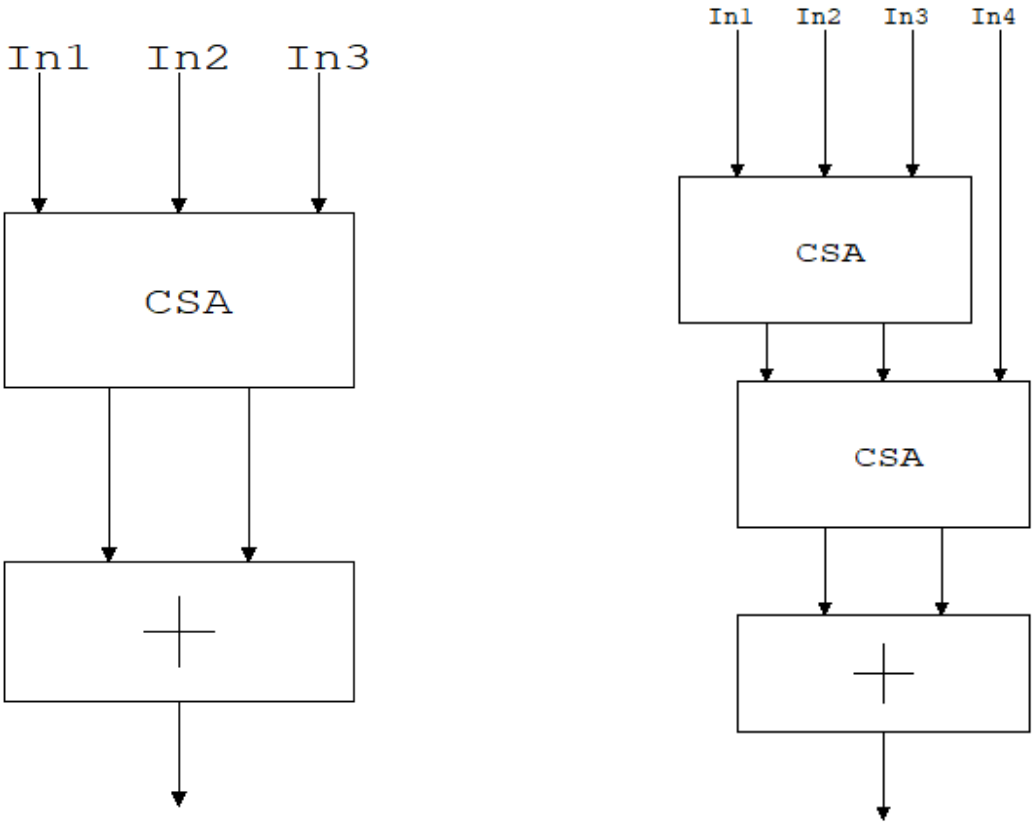

Figure 3.4: Architecture with multi-operand additions schematic

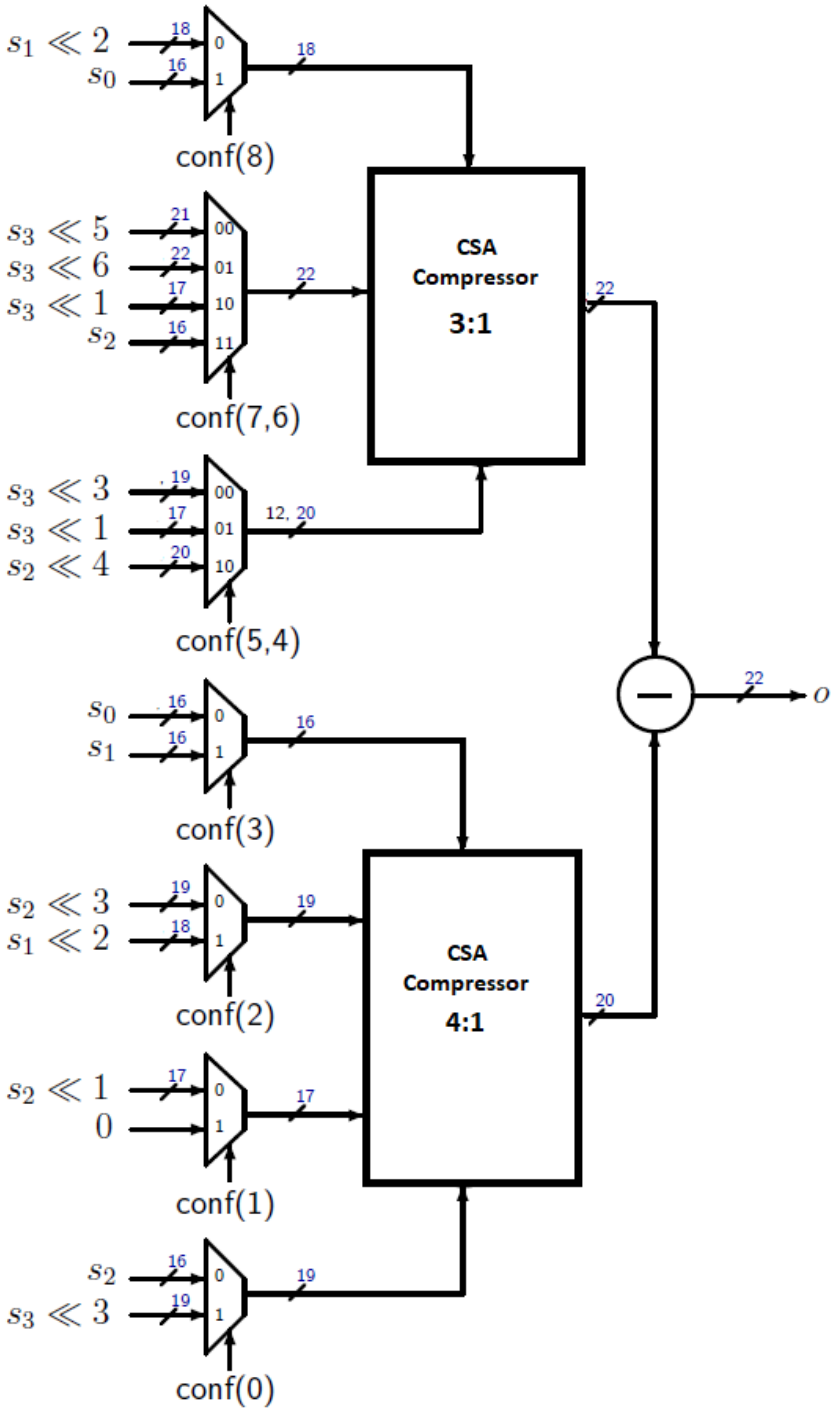

Figure 3.5: Luma Legacy Architecture with CSA

3.4 Generic Accuracy Configurable Adders

An approximate computing approach is applied in order to earn in speed, area and energy efficiency with the drawback of a lower accuracy, exploiting the error tolerance that characterize an application like video-coding. Indeed in this case the output is not required to have the maximum precision, but some approximation can be accepted under certain quality constraints in signal processing.

3.4.1 Complementary Modules in Arithmetic Datapaths

A particular kind of adder shows the feature of accuracy reconfigurability. This particular module is composed of different parts: a precise mode and one or more non-precise modes. An additional Error Detection and Correction (EDC) network is added in order to track the error and to reduce it at the output by selecting consequent appropriate operations.
One possibility consists in the adoption of Complementary Modules [13]: to either cancel or at least reduce the error introduced by the previous approximate operation, a complementary module is chosen in the current addition. This method results efficient because it exploits the arithmetic operation itself to correct the error without any area overhead that can be induced by an additional error correction network. The working principle scheme is represented in figure 3.6.

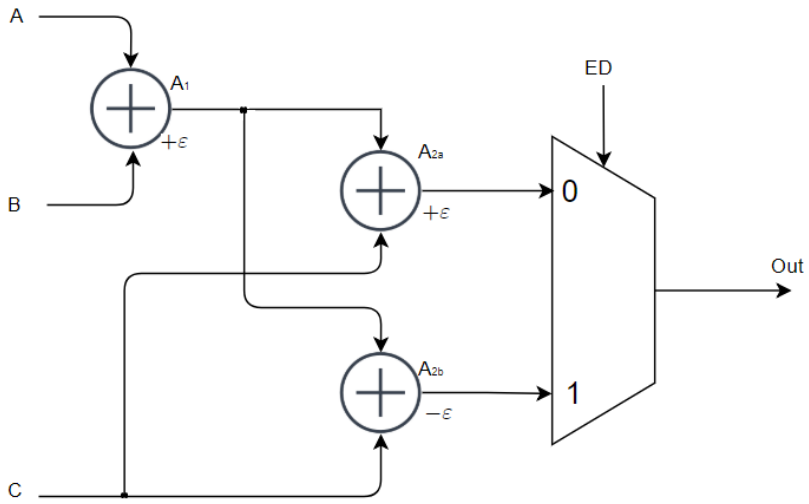

Figure 3.6: Scheme Principle of Complementary Module

The key idea of this network follows this principle: if the approximate adder A_1 commits an error $+\varepsilon$, the addition A_{2b} is selected as subsequent block to decrease this inaccuracy, since it introduces a variation $-\varepsilon$ in principle; otherwise sum A_{2a}, namely SAM (Standard Approximate Module), is employed as successive operation. This selection is allowed thanks to an error detection (ED) signal that is predicted during the first approximate computation. If A_{2b}, namely CAM (Complementary Approximate Module), is chosen, the cumulative error will be partially or completely deleted.

The following elements are needed in the implementation of an adaptive reconfigurable datapath:

- Approximate modules in standard and complementary version: to have errors with same magnitudes and opposite polarities;
- Error Detection (ED) signal: to highlight the necessity of a complementary module in the following operation;
- Mechanism to switch between the standard or complementary approximate component.

One of the main applications of this scheme of principle is the sum-of-products, which implies on operations that are typical of signal processing, as in the concerned environment of this work of thesis.

3.4.2 General Structure

A **Ge**neric **A**ccuracy Configu**r**able (GeAr) [14] adder supports a generic model for block-based adders which exploit multiple sub-adders units of equal length and a error correction unit to provide accurate results when it is required.
Given two N-bits operands to be added, a GeAr computes the sum through k sub-adders that perform the sum operation in parallel. Let R be the number of resultant bits contributing to the final sum and P the number of previous bits used for the carry prediction for each sub-adder. A part from the first one which computes the precise sum over L=R+P ($L \leq N$), all the other sub-adders are R-bit blocks whose carry-in is generated by a Carry Generator Unit. This last consists in a P-bits Carry Look-Ahead adder.
Given a generic N-bit Generic Accuracy Configurable adder, denoted as GeAr(N,R,P), the number of sub-blocks k is computed as:

$$k = \frac{N - L}{R} + 1 \tag{3.2}$$

The architecture of a Generic Accuracy Configurable adder is illustrated in figure 3.7.

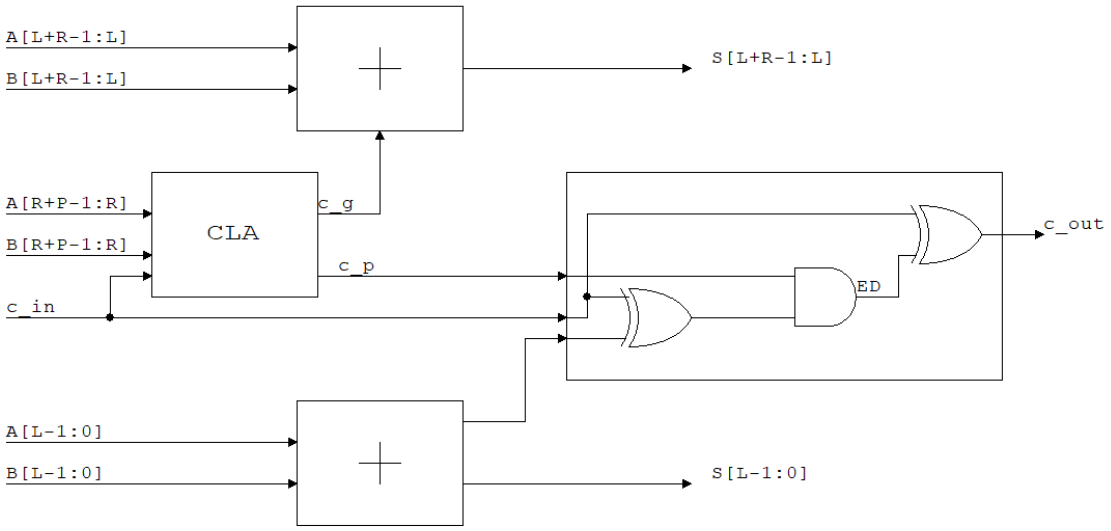

Figure 3.7: Architecture of GeAr and CGeAr, k=2

A Standard GeAr adder discern from a Complementary GeAr (CGeAr) by just the input carry c_{in}. Indeed an excess carry-in is propagated to the next adder in order to compensate the deficiency of the first operation if it provides some error. Since the carry-in of a GeAr is always set to 0 for the MSBs block, it will give an output that is always lower than the correct result; thus in the successive addition this carry will be set to 1, such that the CGeAr will provide a result that is greater than the precise sum, by reducing the overall error.
This is the main improvement of this architecture: the previous inexact sum is in some ways balanced by the current imprecise sum, without the addition of any error correction network.
For what concerns the error detection mechanism, it just requires two additional gates. Indeed the prediction carry bit (cp) is computed by the carry prediction logic of the Carry Look Ahead adder, so there is no additional logic required, through the following formula:

$$cp = \prod_{k=0}^{P-1} A[k] \oplus B[k] \tag{3.3}$$

The expression that is used for the prediction carry takes part in the carry generation process (cg) as propagation term P, following this rule:

$$cg = c_{in,L} = G_{(R,L-1)} + P_{(R,L-1)} \cdot c_{in} \tag{3.4}$$

Therefore the error detection signal can be evaluated for the j^{th} sub-adder as follows:

$$ED_j = cp_j \cdot (cin_j \oplus cout_{j-1}) \tag{3.5}$$

This means that the error detection mechanism needs just additional XOR and AND gates to be estimated.
In particular, an example of the Carry Look Ahead Generator Unit is proposed for P=2, as depicted in figure 3.8, where:

- Generation and Propagation Unit (GPU): computes the generate and propagate bits g_i, p_i starting from a_i, b_i;
- Parallel and Prefix Unit (PPU): provides G, P bits, given g_i, p_i, with a tree-like structure.

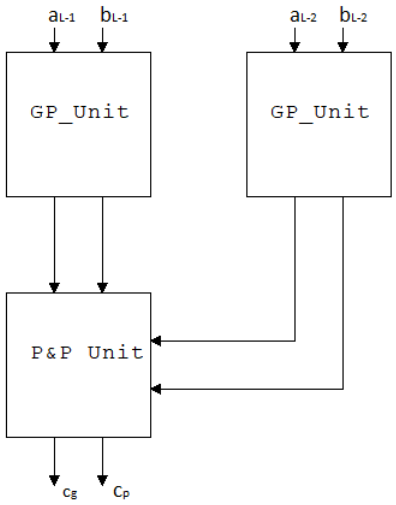

Figure 3.8: Architecture of Carry Generator Unit for P=2

Since CGeAr and GeAr differentiate only for the carry input signal, the logic circuits that change the c_{in} for the i^{th+1} adder will exploit the error detection signal of the previous GeAr block:

$$cin_{i+1,j} = cin_{i,j} \oplus ED_{i,j} \tag{3.6}$$

As a final result, the design of a faster architecture is implemented, with a lower produced error at the cost of a little area overhead.
An alternative to the partition into two different sub-blocks involves three distinct sub-adders based on the same principle as before. The operating scheme is presented in figure 3.9.

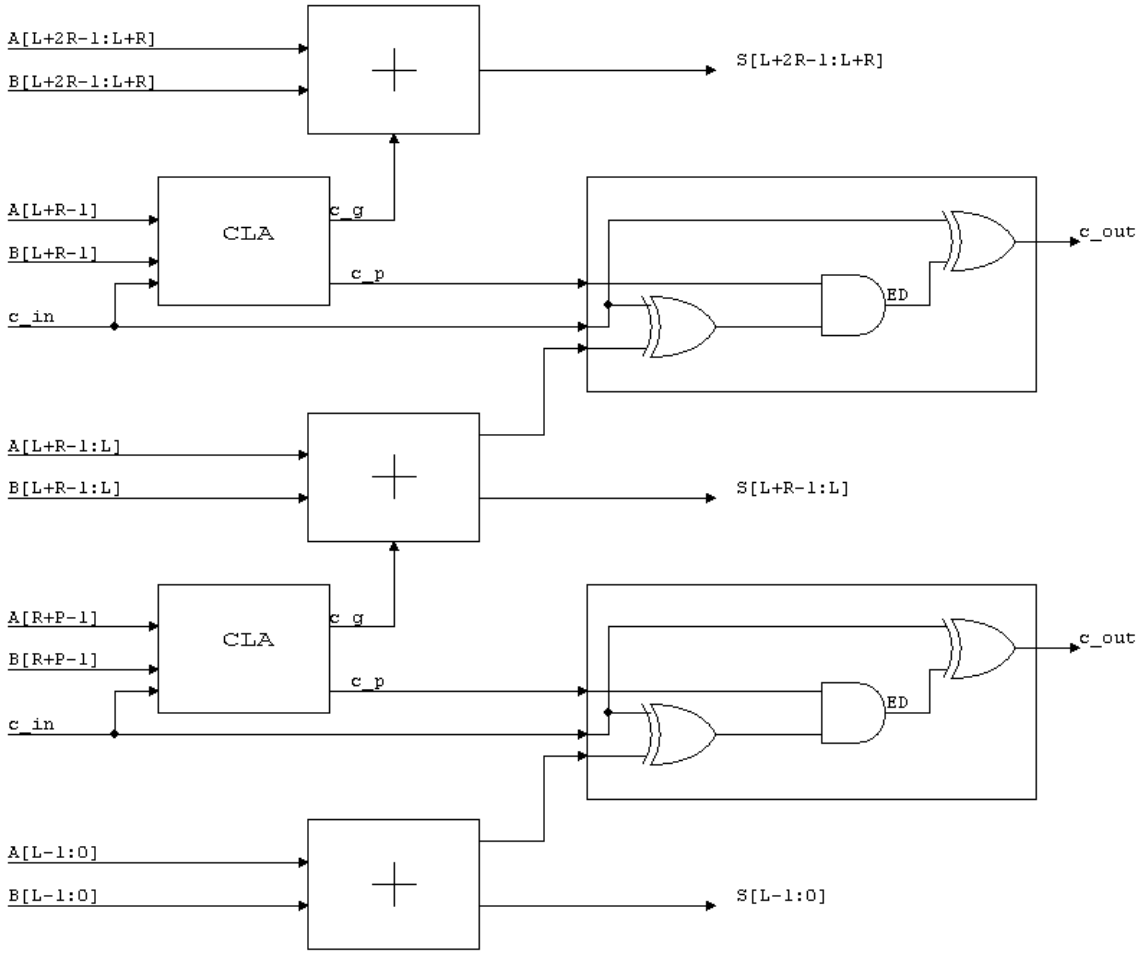

Figure 3.9: Architecture of GeAr and CGeAr k=3, P=0

Since the DCT-IF structure is composed by a tree of adders and not by consecutive sums, the majority of adders are chosen with an adaptive approximate configuration. In particular, four out of five adders are replaced by reconfigurable structures (yellow additions), as presented in figure 3.10.

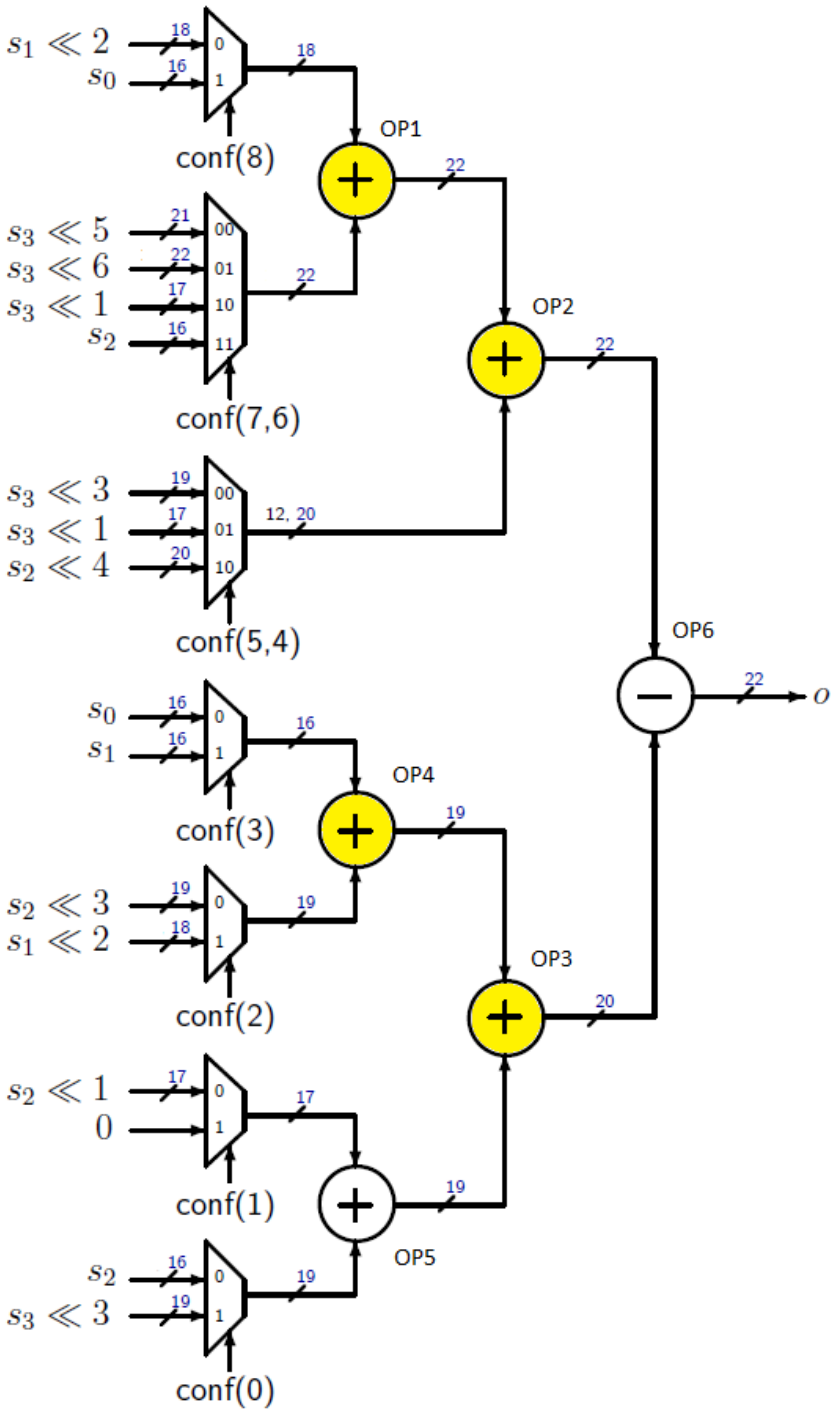

Figure 3.10: Luma Legacy Architecture with adaptive approximate configuration

Two different configurations have been designed with CGeAr, one with P=0, the other with P=2. In particular the P=0 architecture is chosen in order to let the carry propagation chains of different sub-adders as close as possible to each others, in order to fully shorten the critical path. As matter of fact, for a Generic Accuracy Configurable Adder GeAr(22,10,2), while the LSBs L-bits precise sub-adder computes the sum on 12 bits, the R-bit approximate sub-adder performs its operation on 10 bits. On the other hand in a GeAr(22,11,0), the carry propagation paths are exactly the same length. Thus, this will fully enhance the speed performances on the sum computation, with identical paths for the different subcomponents. Moreover, a slightly different configuration is adopted for the case P=0, since in the carry generator unit the Carry Look-Ahead adder is replaced by the carry generation only on the $(L-1)^{th}$ bit, which implies on a lower area overhead:

$$\begin{aligned} cg &= A_{L-1} \cdot B_{L-1} + c_{in} \cdot (A_{L-1} \oplus B_{L-1}) \\ cp &= A_{L-1} \oplus B_{L-1} \end{aligned} \tag{3.7}$$

However for a fixed number of L bits, the greater is P, the lower will be the error probability, with the drawback of a higher power dissipation and occupied area. So it is expected that for a the case P=0, the number of errors is higher even if a shorter critical path is concerned.

3.5 Simulations

3.5.1 Filter Output

Hence, interpolation filters system has been simulated in order to obtain an estimation of the number of errors that are present on a total amount of 36864 samples and to and to evaluate the Root Mean Square Deviation introduced by the approximate adders.

	RMSD	# errors	% errors
HC_Approx	4539.3	1424	3.86%
LF_Approx	13954	4553	12.35%
GeAr k=2, P=0	26.99	28220	76.55%
GeAr k=2, P=2	43.16	20877	56.63%
GeAr k=3, P=0	236.02	34817	94.45%
GeAr k=3, P=2	236.2	33760	91.58%

Table 3.2: Luma Legacy Comparison RMSD, 16 bits out

As expected, since a uniform statistic is taken into account, a higher error rate for parallel & prefix architectures is obtained with respect to the chroma legacy architecture if an approximate approach is adopted for the adders in the filter design.
For what concerns the adaptive reconfigurable architectures that employ the GeAr adders, even if for a lower P a higher number of errors is accomplished, the standard deviation results lower than the case of 2 P-bits. Therefore, the GeAr(N,R,0) will be the one employed for the successive analysis of performances.

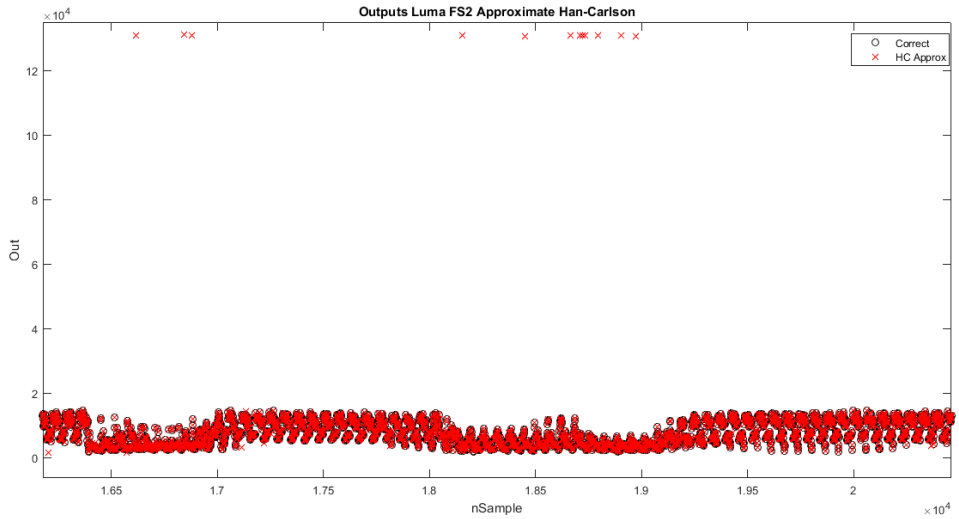

Figure 3.11: Luma Legacy Filter Output, Han-Carlson Approximate Adder topology

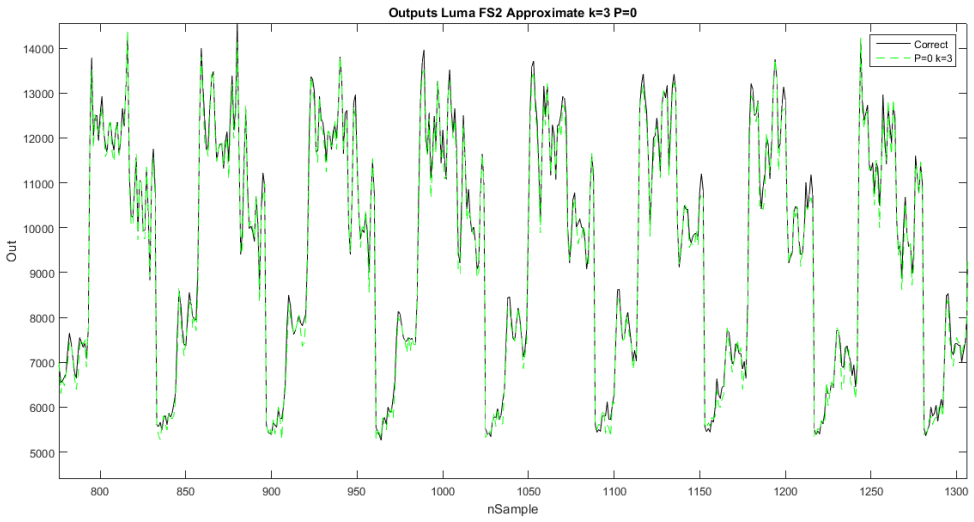

Figure 3.12: Luma Legacy Filter Output, GeAr k=3 P=0

3.5.2 HEVC Output

From the previous section it is noticeable that the architecture which includes Generic Accuracy Configurable adder is the one responsible for the lowest computed RMSD besides it accomplishes the highest percentage of errors, so it could be interesting to verify its effect in terms of PSNR degradation on the entire HEVC system.
In principle the hardware architecture representation must perform exactly the same operations as the proposed software code. For the proposed architecture, this implies on reconfiguring the entire software design which characterizes the luma interpolation process including among the different adders the ones which relate to an adaptive reconfigurable architecture. Since this process is complex and requires a lot of time, an approximation can be performed in order to speed up this modification. If a small enough error ϵ is accomplished on the hardware interpolation filters architecture, the same operation can be obtained by the software model adding a variation ϵ' with the same probability distribution with a negligible difference between the two designs. This will be the procedure applied on this work of thesis, as depicted in figure 3.13.

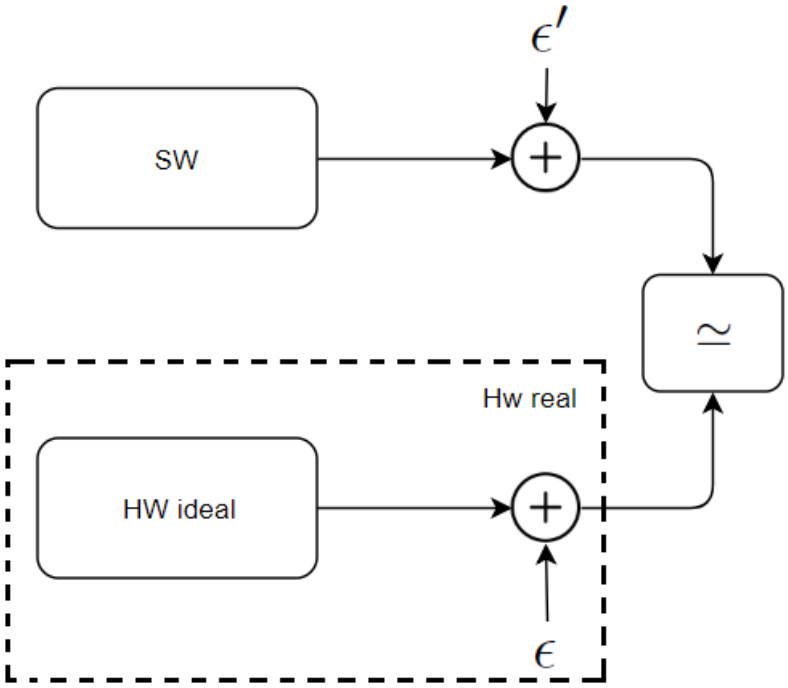

Figure 3.13: Hardware-Software behavior

First of all it is necessary to identify the probability density function that characterizes the interpolation process. The deviation from the correct value is computed per each sample, than its distribution is plotted as histogram. Two possible distributions are taken into account:

- Normal Distribution (bell curve): a continuous probability dispersion to represent random variables characterized by unknown distributions [15]:

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.8}$$

- Logistic Distribution: a continuous probability dispersion with is similar in shape to the normal distribution but with heavier tails:

$$f(x|\mu,\sigma) = \frac{e^{\frac{x-\mu}{\sigma}}}{\sigma\left(1+e^{\frac{x-\mu}{\sigma}}\right)^2} \tag{3.9}$$

The obtained probability density functions for k=2,P=0 are presented in figure 3.14. Fitting data with the object probability function, the logistic distribution shows a mean μ =11.43 and a standard deviation σ=14.14, while the bell curve has μ=12.15 and σ=24.11, which are exactly equal to the statistics of the obtained results. Therefore the normal distribution results a more appropriate choice to model the behavior of the error statistic.

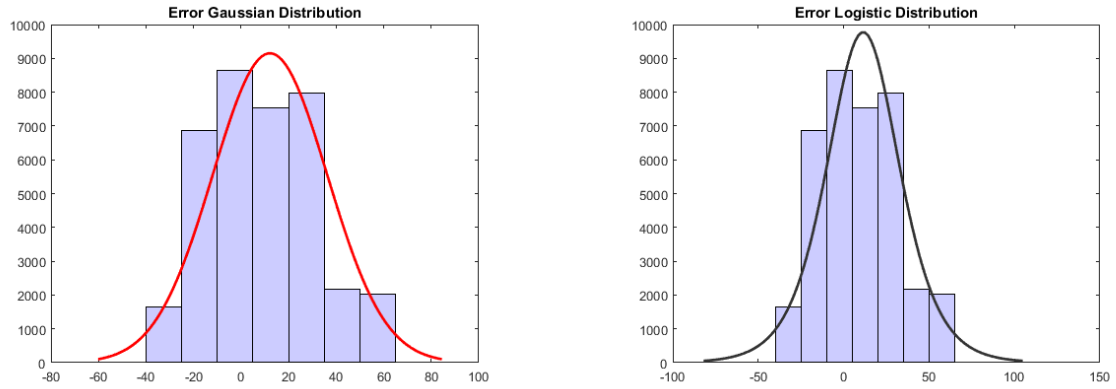

Figure 3.14: Probability Density Function for error distribution k=2, P=0

The probability density functions are evaluated also for the architecture k=3,P=0. Also in this case the Gaussian distribution seems to better approximate the error trend, with the difference that it is composed by the superposition of three normal density functions with similar standard deviation and different mean. The obtained values are presented both in tabular (Tab:3.3, Tab:3.4) and in graphical shape (Fig:3.15).

	Gaussian 1	Gaussian 2	Gaussian 3
μ	-41.09	214.23	472.73
σ	41.98	42.51	41.63

Table 3.3: Mean and standard deviation for Gaussian distributions k=3, P=0

	Logistic 1	Logistic 2	Logistic 3
μ	-39.14	216.27	475.05
σ	25.11	25.41	24.92

Table 3.4: Mean and standard deviation for Logistic distributions k=3, P=0

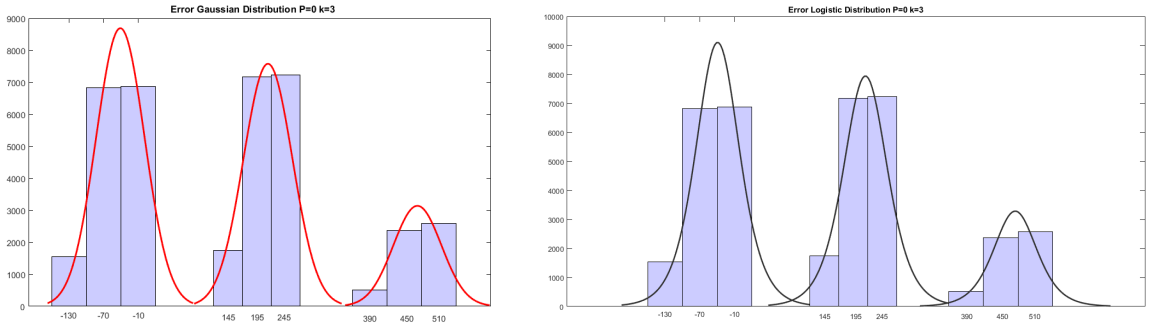

Figure 3.15: Probability Density Functions for error distribution k=3, P=0

Let's analyze these results more in detail, by extracting the error out of each addition that composes the second stage filter. It can be noticed that the deviation from the correct outcome concerns values equal to 16384 and 128 for adders 1 and 2 that compute a 22-bit sum (Fig. 3.16): these results correspond to the positions where the adder is partitioned in different parts. Actually each inexact adder is composed by three sub-units, where the first sum is a precise L-bits addition, while the other two approximate operations compute the output on almost R-bit addends. If N equals 22 and P amounts to 0, it follows that the LSBs are computed up to the bit of weight 2^6, while the intermediate addition involves values from 2^7 up to 2^{13}. Therefore the bits which are mostly affected by an imprecise computation are the ones of weight 2^7 and 2^{14}, as highlighted by the obtained error. Similar results are reached for the error committed by 20-bit adders whose weight is 2^6 and 2^{12}.
These outcomes match at the filter subtraction, whose error values are combinations of numbers that are all multiples of 2. Since the Luma architecture consists in two mirrored stages of filters, their outputs are summed up in a final addition. Hence the last operation consists in shifting the output of 6 right positions, thus decreasing the weight of the obtained deviation from 2^{15} to 2^9 and so on. This justifies the three gaussian curves.

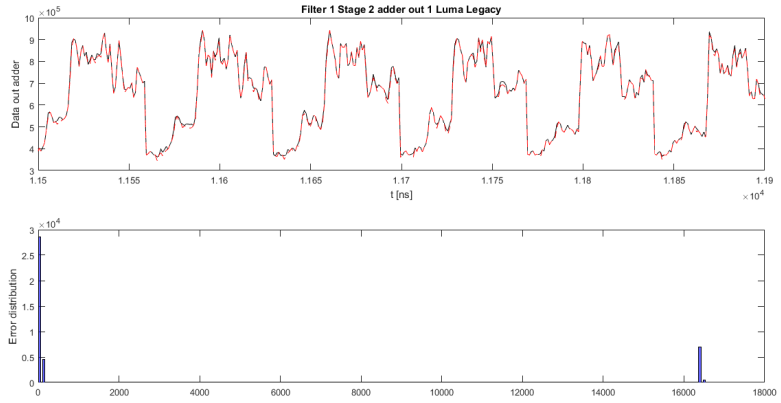

Figure 3.16: Data statistic adders 1 and 2

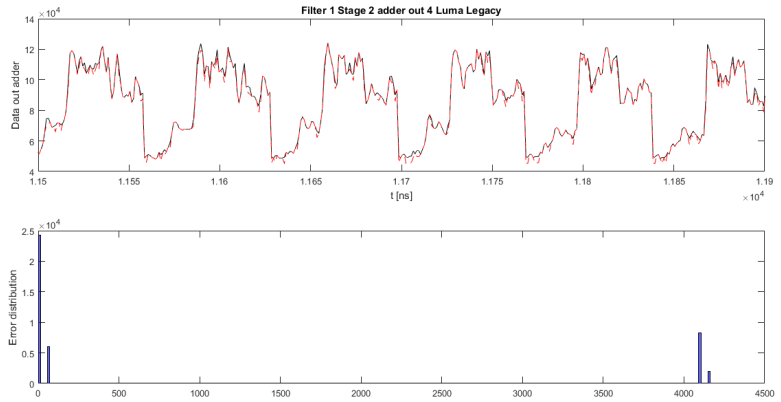

Figure 3.17: Data statistic adders 3 and 4

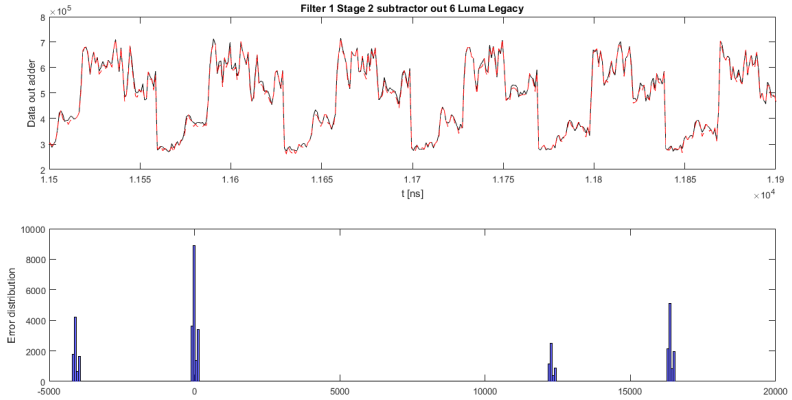

Figure 3.18: Data statistic subtractor 6

Moreover a random noise is generated following the three Gaussian statistics of figure 3.15 and random permutations are performed to make it more reliable with respect to a real random statistic. This deviation is added to the correct outputs of the interpolation filters architecture which are compared to the one obtained by the system with configurable adders: the Root Mean Square Deviation, the Mean Square error (Eq:3.10) and the maximum error with respect to the precise outcomes are evaluated. As reported in table 3.5, the obtained deviation adding noise seems not to distance the one achieved with the approximate architecture.

$$MSE = \frac{\sum_{i=1}^{n}(x_{i,correct} - x_{i,cut})^2}{n} \tag{3.10}$$

	Correct+Noise	GeAr Architecture
RMSD	235.22	236.02
MSE	55330	55707
Max Error	608.65	516

Table 3.5: Error Figures of Merit with gaussian noise

Therefore the model with the noise composed by three Gaussian distributions represents reasonably well the behavior of the architecture with the approximate adders. This error is inserted in the HM software [6] in order to have a more precise idea of the effect that this reconfigurable architecture can have one the PSNR of HEVC. An analysis of the coding efficiency of the entire HEVC system is executed basing on the same concepts of section 2.2.3. The noise is introduced in the HM code pursuing the following strategy: three Gaussian noises, each one with its mean and standard deviation as in table 3.3, are generated through the Box-Muller algorithm with a seed different from zero. Then they are merged into a single noise vector which is added to the filtered data, provoking a distortion from the correct behavior.

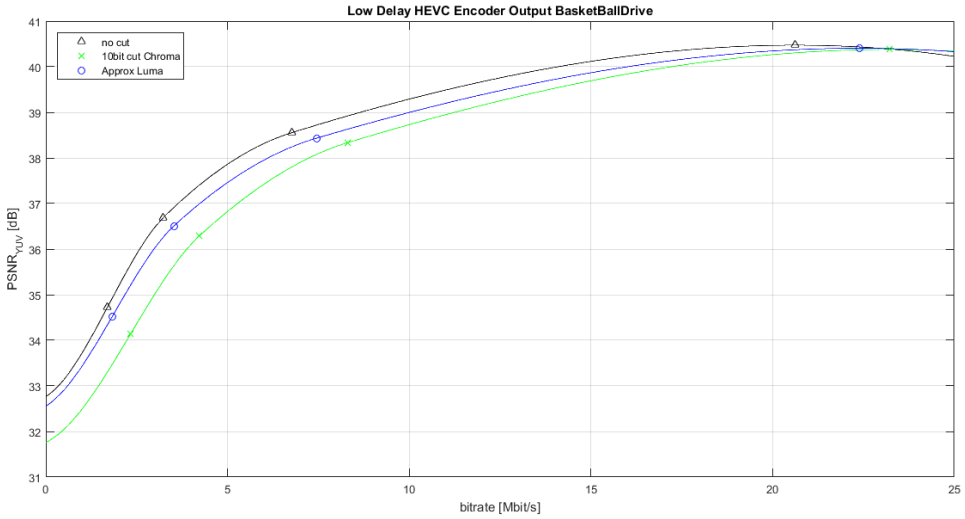

Figure 3.19: PSNR degradation with Luma approximate computing HEVC encoder (BasketballDrive[17], 1920x1080, 50 Hz, Low Delay)

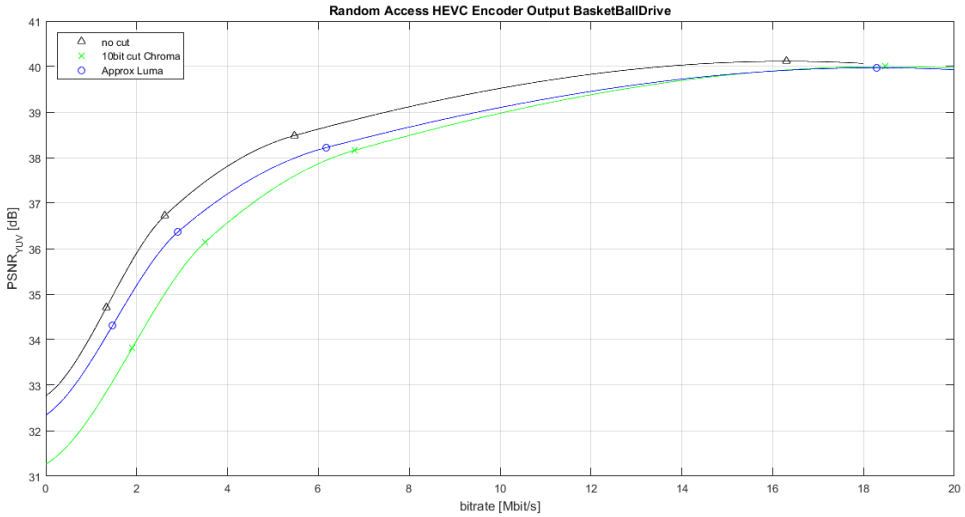

Figure 3.20: PSNR degradation with Luma approximate computing HEVC encoder (BasketballDrive[17], 1920x1080, 50 Hz, Random Access)

As highlighted in figures 3.19 and 3.20, the obtained PSNR curve with generic accuracy reconfigurable adders shows a limited distortion with respect to the correct architecture. In particular the PSNR degradation is less than a dB for higher QP (-q 37, -q 32), while it increases when a lower QP (-q 22) is involved, so small variations are more likely captured. Moreover it can also be noticed that the GeAr architecture presents a less pronounced distortion when the Low Delay prediction structure is employed.
For the sake of completeness the PSNR that is achieved with the previous Chroma cut architecture is reported in figure. It is noticeable that the PSNR with GeAr adders introduces a Signal-to-Noise-Ratio that is way more higher with respect to the previous cut architecture.
It is remarkable to notice that the obtained analysis is carried out as a worst case evaluation for the proposed architecture. Indeed the three Gaussian distributions are achieved through Modelsim simulations that neglect the 1D interpolation, whose results are always correct, since the second stage approximate filter is not involved. This assumption is not applied to the HEVC reference software, obtaining a PSNR distortion that is larger than the one that will be recovered in reality. As a conclusion, the obtained results show that the PSNR with the proposed architecture is representative that an acceptable distortion is introduced in the original interpolation filters architecture, resulting in tolerable approximate results as output of the entire HEVC system if both an approximate encoder and decoder are employed.
Furthermore a complete analysis of the HM system is carried out for all the possible configurations with the proposed architecture. In addition to the entire correct and approximate architecture, two hybrid conditions are added (correct encoder and approximate decoder and vice-versa) in order to explore all the possibilities. In particular, three different sequences which belong to distinct classes (because of disparate resolution) are analyzed: *BasketballDrive.yuv* (Class B, 1920x1080, 500 Frames, 50 Hz frame rate), *BasketballDrill.yuv* (Class C, 832x480, 500 Frames, 50 Hz frame rate), *RaceHorses.yuv* (Class D, 416x240, 300 Frames, 30 Hz frame rate).

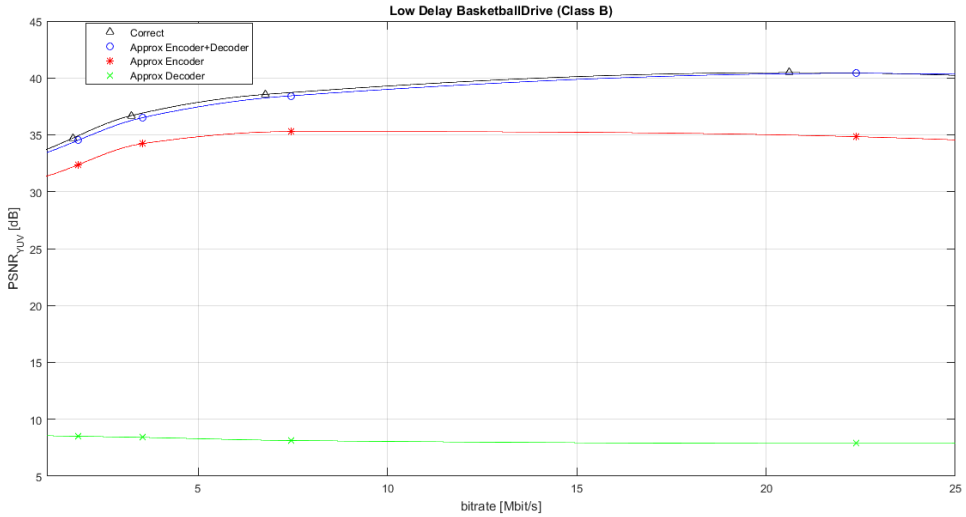

Figure 3.21: PSNR degradation Encoder-Decoder Combinations (BasketballDrive[17], 1920x1080, 50 Hz, Low Delay)

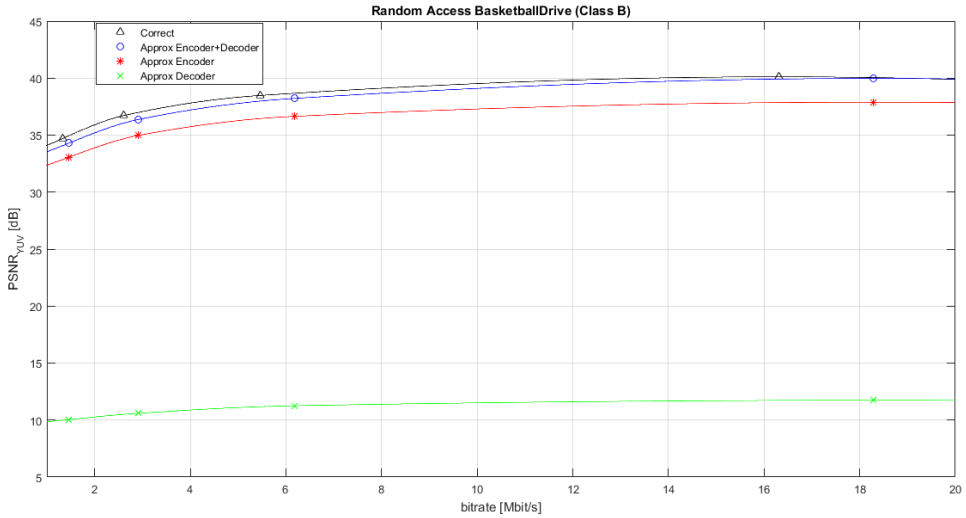

Figure 3.22: PSNR degradation Encoder-Decoder Combinations (BasketballDrive[17], 1920x1080, 50 Hz, Random Access)

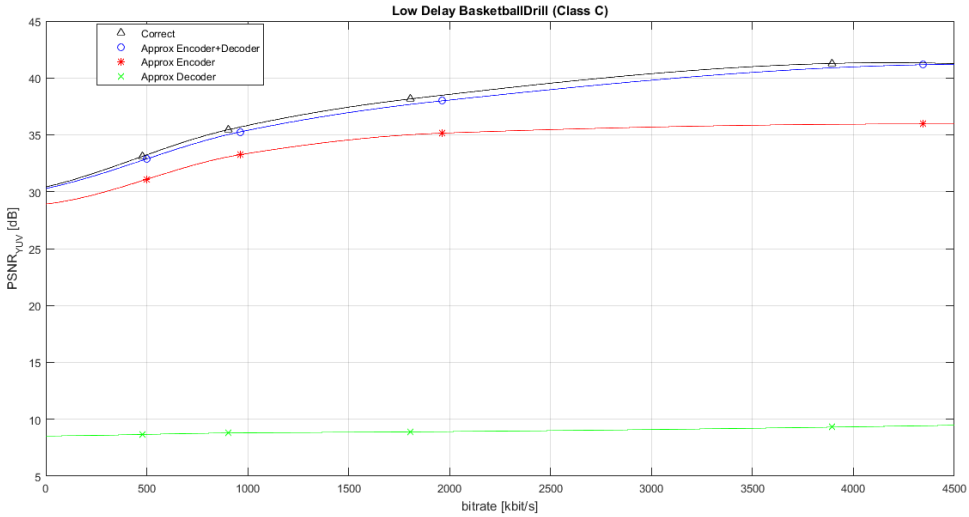

Figure 3.23: PSNR degradation Encoder-Decoder Combinations (BasketballDrill[17], 832x480, 50 Hz, Low Delay)

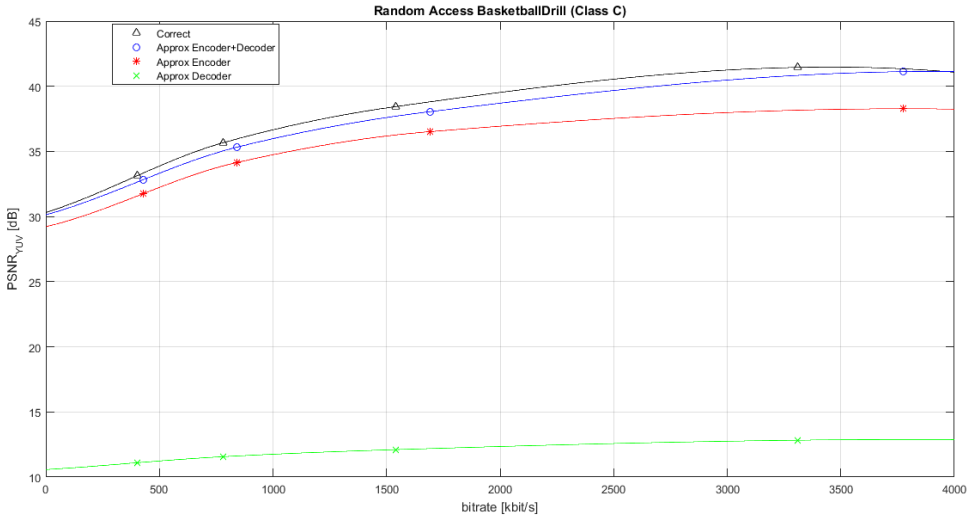

Figure 3.24: PSNR degradation Encoder-Decoder Combinations (BasketballDrill[17], 832x480, 50 Hz, Random Access)

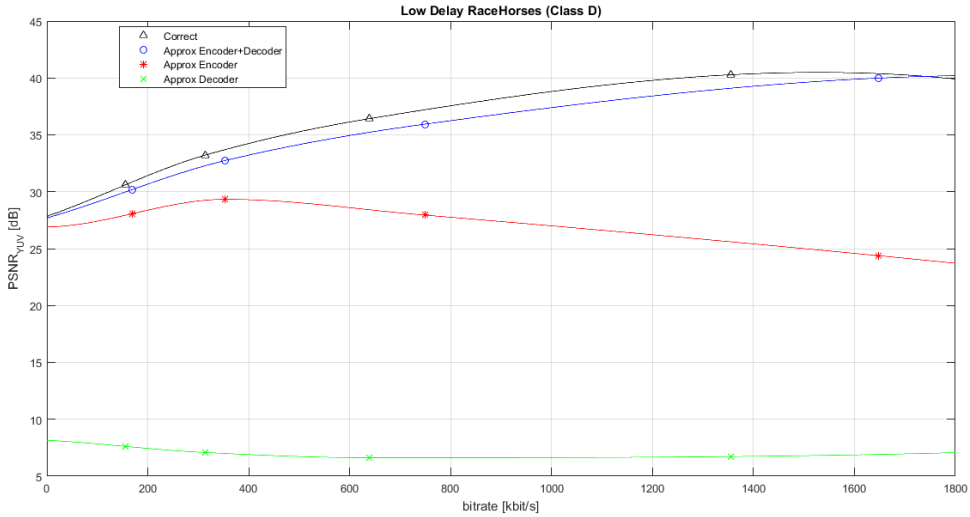

Figure 3.25: PSNR degradation Encoder-Decoder Combinations (RaceHorses[17], 416x240, 30 Hz, Low Delay)

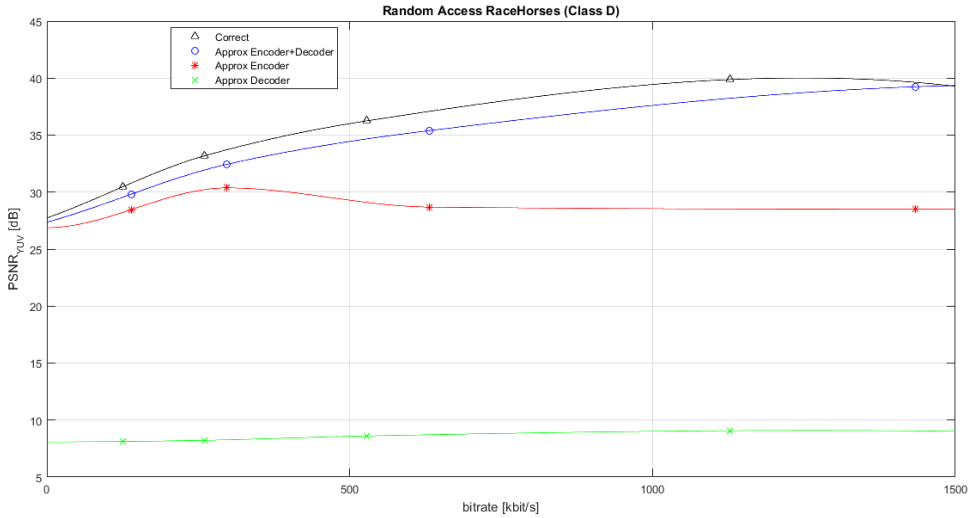

Figure 3.26: PSNR degradation Encoder-Decoder Combinations (RaceHorses[17], 416x240, 30 Hz, Random Access)

As shown in figures, it could be concluded that if just an approximate encoder is applied, the PSNR gets worse especially for a higher bitrate, while performances degrade in a significant way if an approximate decoder is employed. All in all, the solution with approximate encoder and decoder results the best choice for an acceptable PSNR as output of the entire system.

3.6 Design Synthesis Results

The different interpolation filters structures with the proposed adders architectures have been synthesized with the same conditions as the Chroma Legacy case, as presented in section 2.3.5. Also in this case the clock-gating technique is employed in order to enhance the overall performances of the system.

	f_{max} [MHz]	Area [μm^2] [2]	Power [mW] [2]
PE correct	588.24	57166.92	9.950
PE HC correct	591.72	60995.88	11.652
PE HC approx	588.24	63935.28	12.706
PE LF correct	595.24	58959.36	11.254
PE LF approx	588.24	62618.40	12.261
PE CSA	588.24	58930.56	11.092
PE GeAr P=0 k=2	595.24	59286.96	11.115
PE GeAr P=0 k=3	602.41	57878.28	10.589
PE GeAr P=2 k=3	591.72	61360.92	11.394

Table 3.6: Luma Legacy Filter Synthesis results with clock gating

As far as parallel & prefix adders are concerned, none among the proposed architecture seems to be the best choice for the case of the Luma Legacy Architecture. Beside the Ladner-Fischer correct topology outperforms its competitors in all the characteristics, the drawback in terms of power dissipation is not worth to be used.
Regarding the architecture with Carry save adders, it results unfavorable, because it provides the same maximum frequency as the original solution, with the drawback of a higher area overhead and power consumption.
If an approximate approach with generic accuracy configurable adders is employed, it ensues to obtain better performances with respect to the previous techniques. In particular the Processing Element with GeAr structures characterized by P=0, k=3, shows the most suitable outcomes with an improvement in performances of 2.41% with negligible drawbacks in terms of area overhead (1.24%) and power consumption (6.42%). Therefore this approximate architecture will be the one chosen as new reference structure, since it allows to worth performances with an acceptable accuracy with respect to the correct case.

	Δf_{max} [%]	ΔA [%]	ΔP [%]
PE HC	+0.59	+6.70	+17.11
PE HC App	+0.00	+11.84	+27.70
PE LF	+1.19	+3.14	+13.11
PE LF App	+0.00	+9.54	+23.23
PE CSA	+0.00	+3.09	+11.48
PE GeAr P=0 k=2	+1.19	+3.71	+11.71
PE GeAr P=0 k=3	+2.41	+1.24	+6.42
PE GeAr P=2 k=3	+0.59	+7.34	+14.51

Table 3.7: Luma Legacy relative percentage comparisons with the original design

[2] Area and Power are evaluated at t_{ck}=1.70 ns, that is the worst case for the entire interpolation filters architecture, i.e. luma legacy structure

Chapter 4

Approximate Computing on DCT-IF architecture

The approximate architecture has been introduced in order to reduce the computational complexity of interpolation filters, by reaching a good trade-off between a good quality of video-coding and energy consumption. The number of taps is dynamically reduced as presented in [2], to succeed in this purpose. Indeed a lower order for the filters consists both in a reduced power dissipation and in a higher maximum frequency, with the drawback of a lower precision in the data out of the interpolation process. The same adders topologies as presented in the previous sections are applied to the approximate architectures in order to enhance performances.

4.1 General Structure

Three different solutions are analyzed and their effects are evaluated at the output of the entire HEVC system:

- **Approximate HEVC Decoder**: since the fractional sample interpolation represents the most expensive block in terms of computation at the decoder side, an approximate solution is proposed as depicted in figure 4.1. In this case the obtained PSNR is substantially lower than the legacy structure;

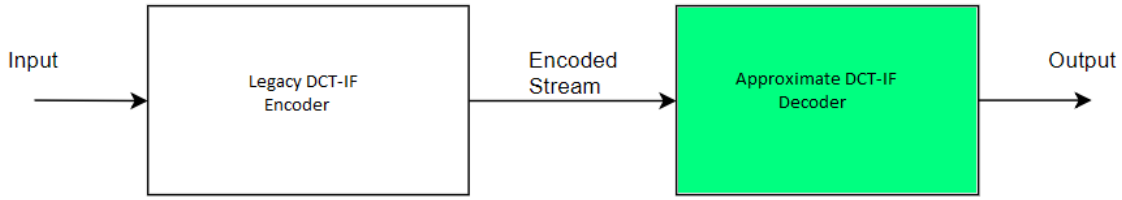

Figure 4.1: Approximate DCT-IF on decoder side

- **Approximate HEVC Encoder**: a lower accuracy approach is also applied at the encoder side, because interpolation filters show a relevant percentage in this part of the HEVC process, too. Here the analysis has been carried out for both random-access and low-delay configurations: for the latter the approximate approach could not be a reliable choice because a too high degradation is obtained at the output of the system.

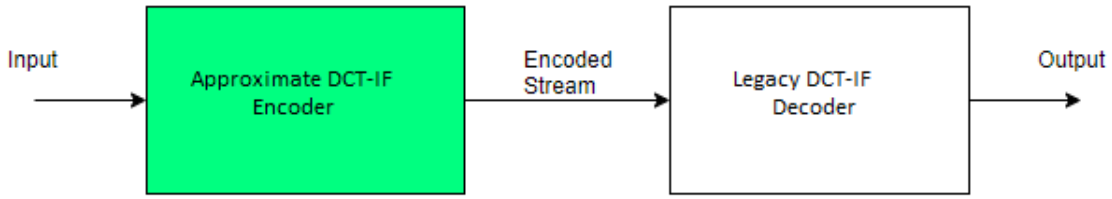

Figure 4.2: Approximate DCT-IF on encoder side

- **Approximate HEVC Encoder & Decoder**: as for the generic accuracy configurable adders in the luma legacy structure, this solution results the most convenient to be applied, since it allows to achieve a lower degradation than the previous two case, with acceptable values in the PSNR evaluation for the entire architecture.

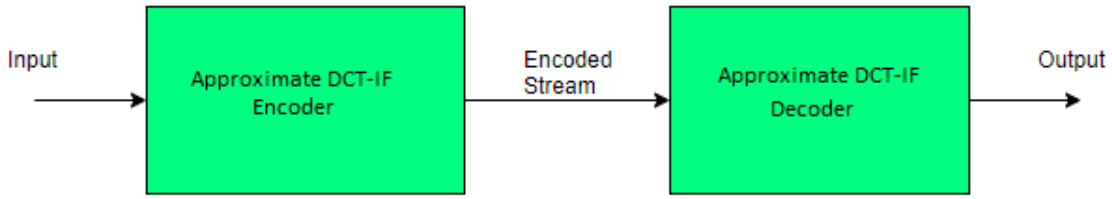

Figure 4.3: Approximate DCT-IF on both encoder and decoder side

The three different solutions are simulated through the HM software [6], whose results are reported in figures 4.4 and 4.5:

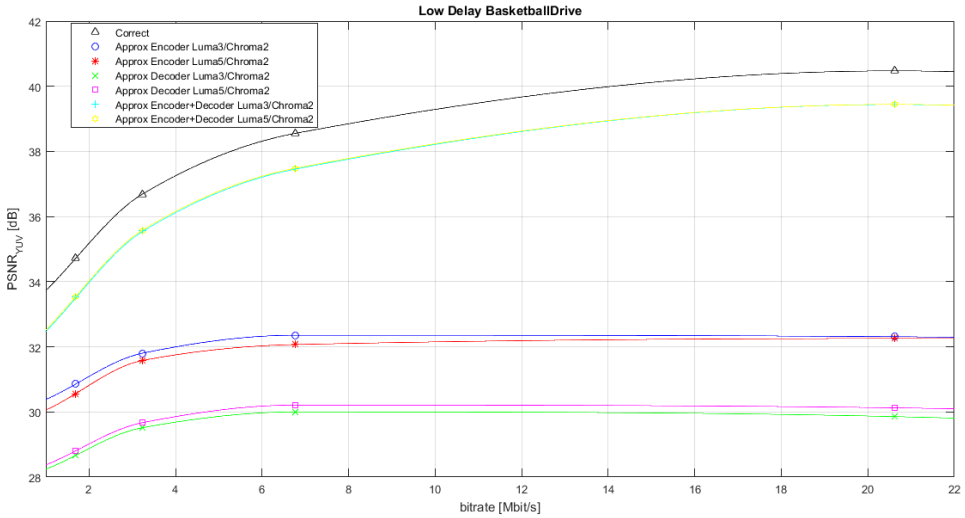

Figure 4.4: Approximate DCT-IF PSNR degradation (BasketballDrive[17], 1920x1080, 50 Hz, Low Delay)

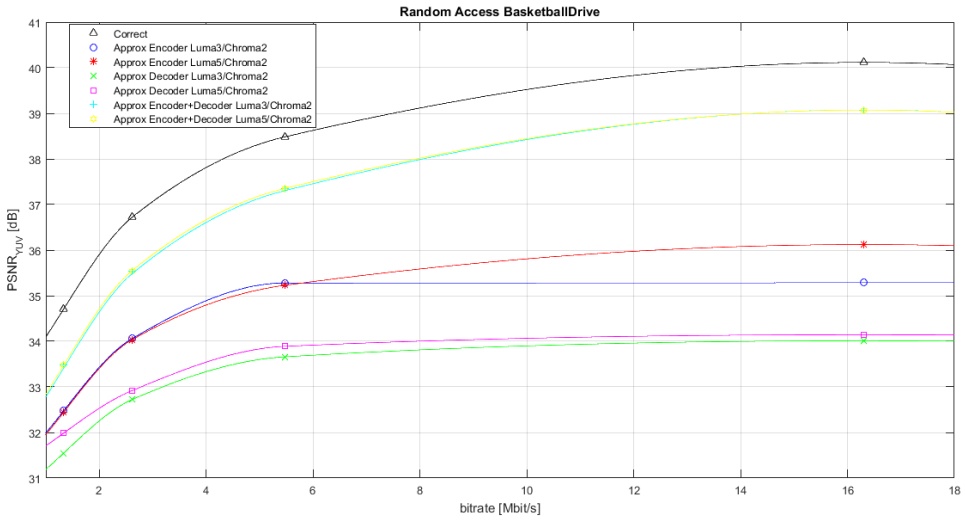

Figure 4.5: Approximate DCT-IF PSNR degradation (BasketballDrive[17], 1920x1080, 50 Hz, Random Access)

4.2 Hardware Design

An architecture that exploits reconfigurable filters is adopted, as proposed in [2]. In particular the 7/8-taps filters of the luma legacy architecture are replaced by 5-tap and 3-tap filters, while the 4-tap filters of chroma legacy structure are flanked by 2-tap components. New coefficients are defined and this results in different sum and shift operations for the DCT-IF structure, as presented in table 4.1.

shifts | coeff	**1**	**2**	**4**	**5**	**6**	**7**	**9**	**14**	20	23	32	40	41	48	50	54	57
x	+			+		-	+			-			+				+
x « 1		+			+			-							+	+	
x « 2			+	+	+				+							+	
x « 3						+	+			-		+	+				-
x « 4								+	+					+	+	+	
x « 5										+	+	+	+	+	+	+	
x « 6																	+

Table 4.1: Approximate architecture Coefficients replaced by sums and shifts

The datapath changes with respect to the legacy architecture: above all, several parallel filter branches are presented, each one associated to a specific reconfigurable DCT-IF implementation. If a filter should not be used in the computation, its input is set to zero by the referring routing unit, in order to avoid the employment of demultiplexers as input of these lasts. The FSMs aren't significantly changed from the starting legacy architecture. The overall datapath for the luma architecture is shown in picture 4.6. Also in this case a multiplier-less solution for the interpolation filters is adopted.

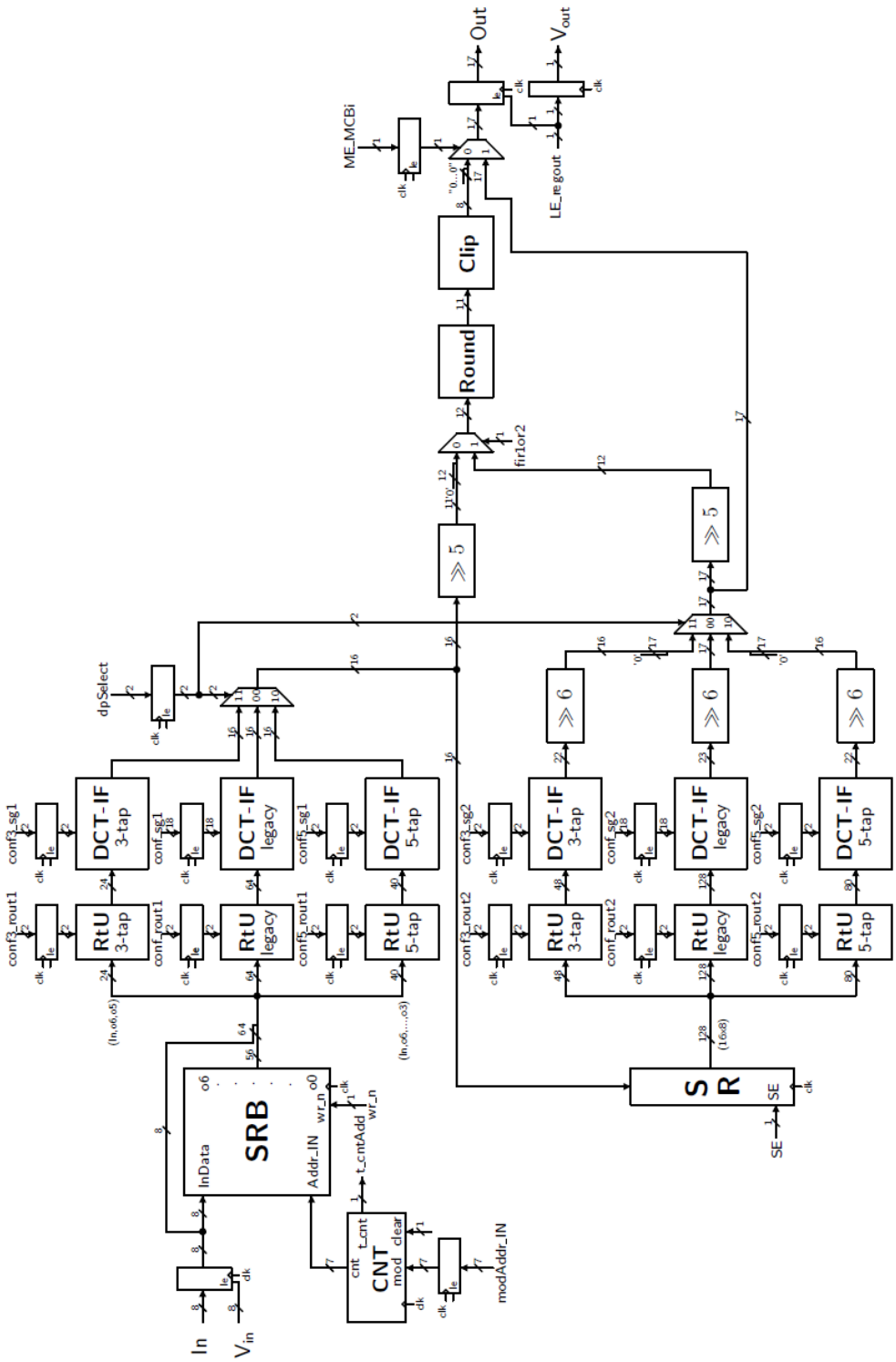

Figure 4.6: Datapath Luma Approximate [2]

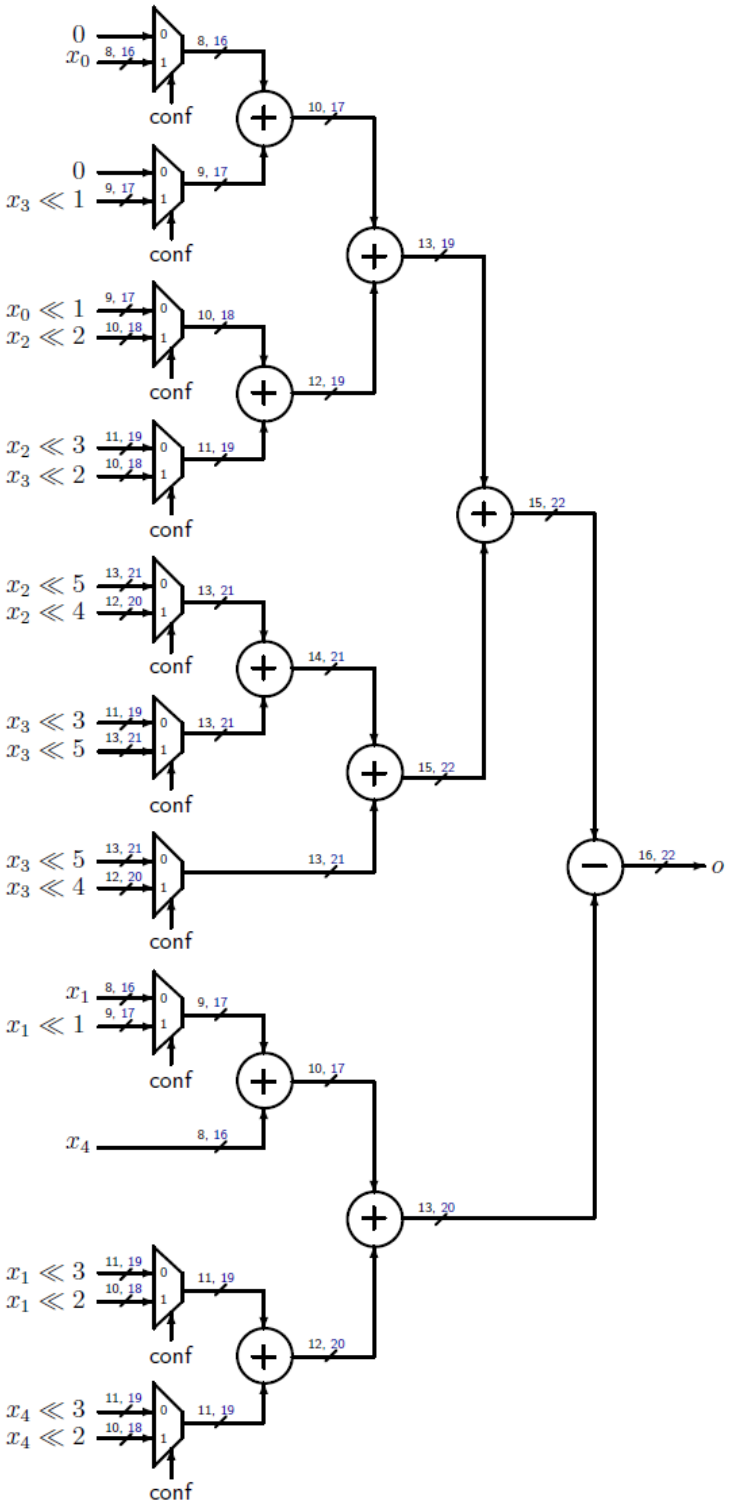

Figure 4.7: Reconfigurable approximate luma 5-tap filter [2]

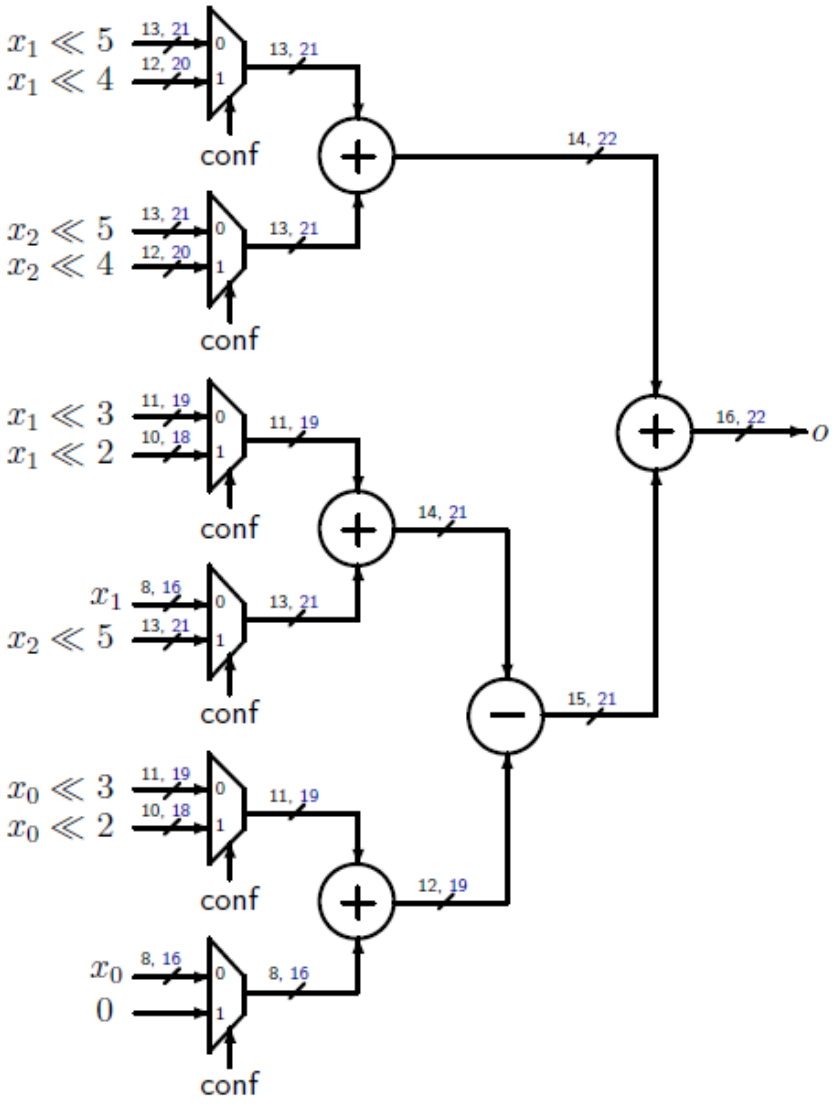

Figure 4.8: Reconfigurable approximate luma 3-tap filter [2]

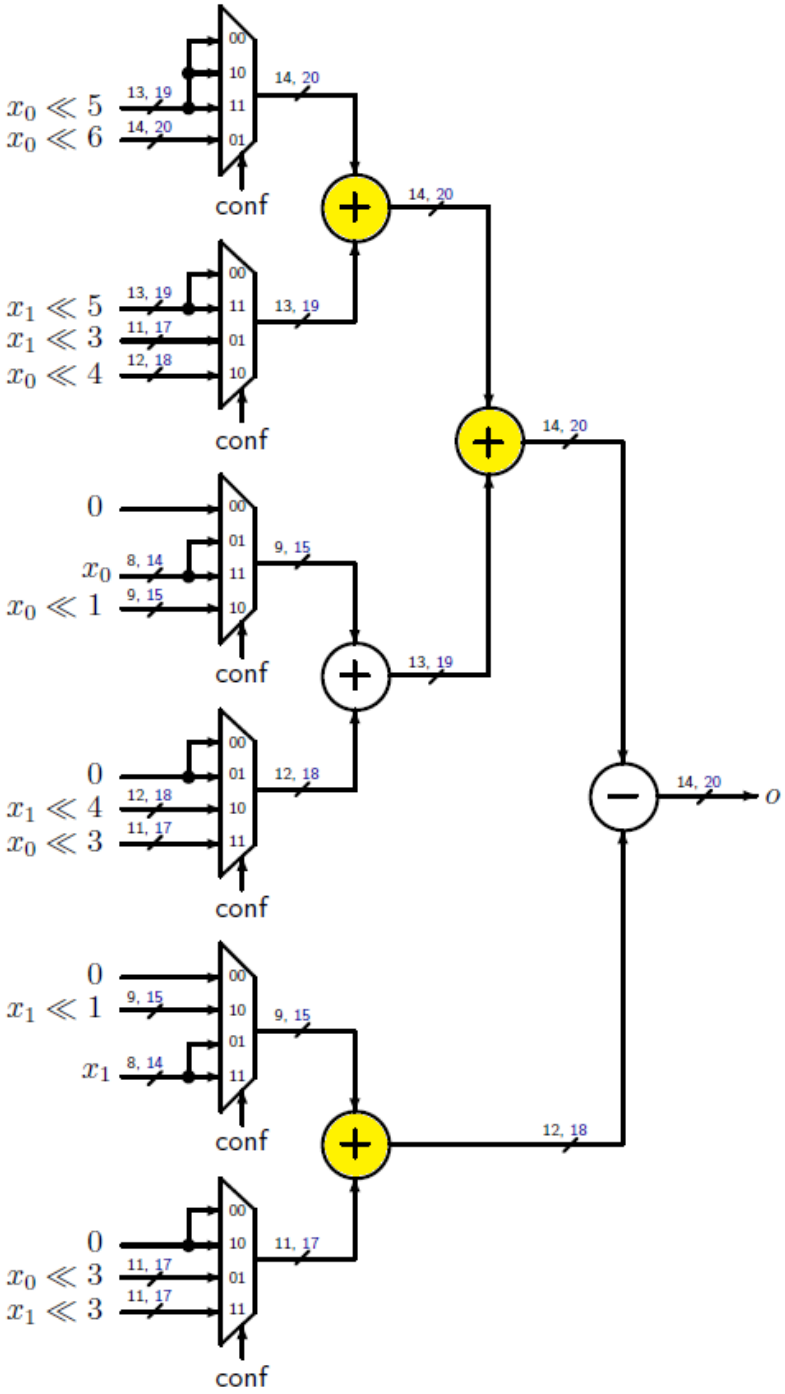

Figure 4.9: Reconfigurable approximate chroma 2-tap filter with GeAr [2]

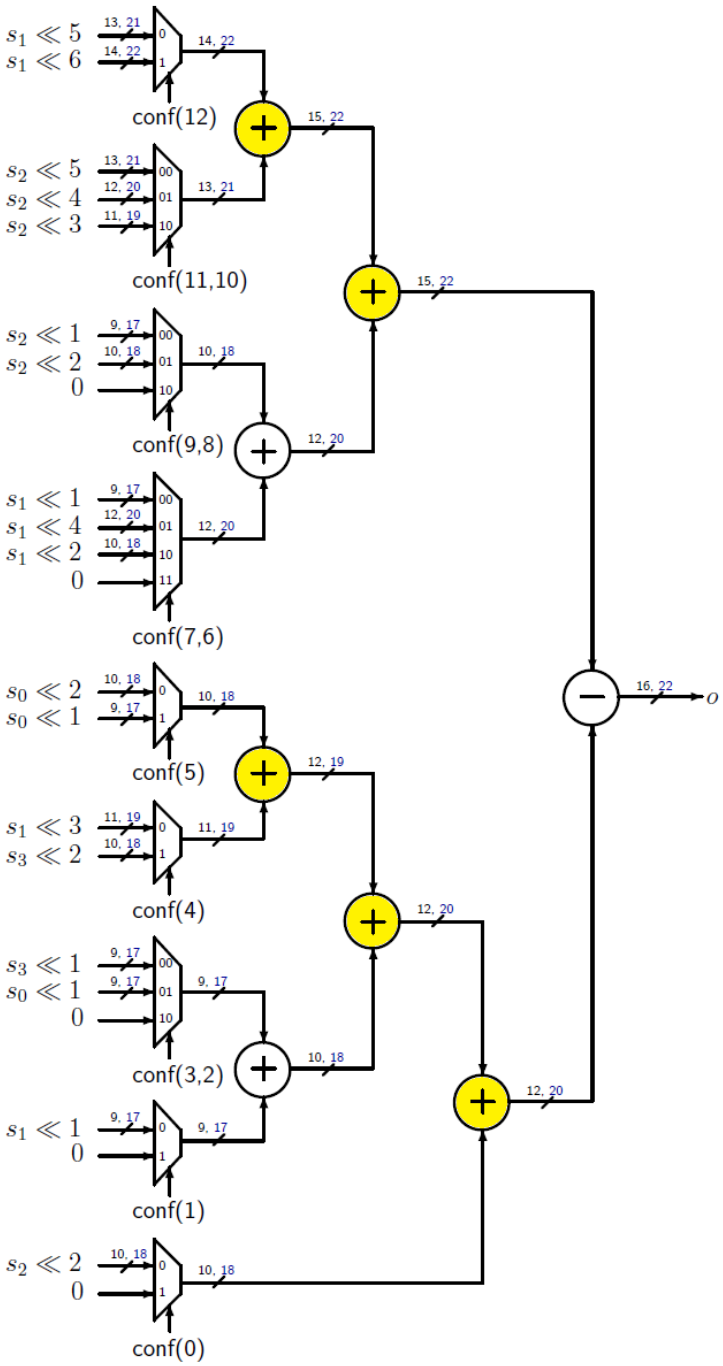

Figure 4.10: Reconfigurable approximate chroma 4-tap filter with GeAr [2]

4.2.1 Adders Topologies

As in the legacy architectures, the different proposed adders topologies have been applied to the approximate DCT-IF structure. For what concerns the chroma approximate, both parallel and prefix solutions and complementary modules with generic accuracy configurable adders are handled to improve performances. Regarding the luma approximate only the parallel and prefix solutions are involved in performances enhancement. For both the architectures both in the legacy part and in the reduced order tap one the original adders are replaced with the proposed ones.
The parallel and prefix adders take the place of all the involved sums in the filtering process, while the GeAr adders replace the majority of the interested additions, represented as yellow circles in figures 4.9, 4.10.

4.3 Simulation

All the different input files have been simulated one after the other in a single simulation, in order to make a rough estimation of the total RMSD and of the number of errors that are present on a total amount of samples (7168 for Chroma Approximate Architecture, 36864 for the Luma one) and to compare the different versions of approximate adders. Simulation results are presented in table 4.2 for the chroma case: as before, the Generic Accuracy Configurable Accuracy adders outperform the Parallel & Prefix approximate solutions in terms of precision.

	RMSD	# errors	% errors
HC_Approx	48.4	1	0.01%
LF_Approx	998.54	4553	63.52%
GeAr k=2, P=0	2.982	249	3.47%
GeAr k=3, P=0	33.372	1949	27.19%

Table 4.2: Comparison RMSD Chroma Approximate Architecture 2-tap, 16 bits out

Tables 4.3 and 4.4 present the accuracy results for the luma case:

	RMSD	# errors	% errors
HC_Approx	670.52	320	0.87%
LF_Approx	1838.4	1291	3.50%

Table 4.3: Comparison RMSD Luma Approximate Architecture 3-tap, 16 bits out

	RMSD	# errors	% errors
HC_Approx	2449.5	1305	3.54%
LF_Approx	4500.8	4148	11.25%

Table 4.4: Comparison RMSD Luma Approximate Architecture 5-tap, 16 bits out

For what concerns parallel and prefix solutions, the inexact architectures produce a low percentage of errors with respect to the total amount, however it's still a high deviation with respect to the original architecture.

4.4 Design Synthesis Results

Approximate DCT-IF interpolation filters are more efficient in terms of energy and power consumption than the legacy ones. The proposed architectures are synthesized using the same power saving techniques (clock gating and switching activity reduction schemes). As reported in table 4.5, as the approximated structure is faster than the legacy one, the power dissipation is reduced. Concerning the luma case, a -8.92% and a -25.8% of power reduction is obtained with the 5-tap or the 3-tap filter respectively. The same behavior is obtained with the chroma processing element, where a -27.3% power reduction is achieved using a 2-tap filter instead of the legacy 4-tap one.

	Power [mW]	$\Delta\%$
Luma Legacy	9.950	–
Luma 5-tap	9.062	-8.92%
Luma 3-tap	7.384	-25.8%
Chroma Legacy	2.966	–
Chroma 2-tap	2.157	-27.3%

Table 4.5: Power results with approximate DCT-IFs compared to the legacy DCT-IFs

The different interpolation filters structures with the proposed adders architectures have been synthesized with the same conditions as the Legacy case, as presented in section 2.3.5, with the employment of the clock gating technique.
Regarding the Chroma Approximate synthesis results none among the explored solutions with Parallel & Prefix architecture shows any improvements with respect to the original proposed architecture, as presented in table 4.6. Therefore, for what concerns these adders, neither Han-Carlson/Ladner-Fischer nor Generic Accuracy Reconfigurable topology are able to reach better results and these solutions are discarded in this case.

	f_{max} [MHz]	Area [μm^2] [3]	Power 2-tap[mW] [3]
PE correct	684.93	17183.52	2.157
PE HC correct	675.68	18917.64	2.412
PE HC approx	675.68	19707.12	2.379
PE LF correct	680.27	18700.92	2.323
PE LF approx	675.68	19126.08	2.354
PE GeAr k=2 P=0	662.25	17974.80	2.314
PE GeAr k=3 P=0	684.93	17621.28	2.225

Table 4.6: Chroma Approximate Synthesis results with clock gating

Results for the Luma Approximate architecture are reported in tables 4.7 and 4.8.

[3] Area and Power are evaluated at t_{ck}=1.70 ns, that is the worst case for the entire interpolation filters architecture, i.e. luma legacy structure

	f_{max} [MHz]	Area [μm^2]	Power 3-tap [mW] [3]	Power 5-tap [mW] [3]
PE correct	591.7	64017.00	7.384	9.062
PE HC correct	595.2	63670.32	7.057	9.131
PE HC approx	602.4	66632.04	7.475	10.167
PE LF correct	602.4	66105.36	7.468	9.391
PE LF approx	595.2	66482.64	7.461	9.725

Table 4.7: Luma Approximate Synthesis results with clock gating

After having synthesized the different architectures with Synopsys Design Compiler, the main advantages both in terms of speed and concerning power and area are obtained through the Han-Carlson correct architecture. Beside this adder topology is not providing the best speed enhancement, it is the one and only that improves frequency performances with a lower area overhead and a remarkably reduced power consumption when a 3-tap interpolation is required. Moreover it is noticeable to observe that this is a correct design, so the enhancement in performances is obtained without any loss in precision.

	Δf_{max} [%]	ΔA [%]	ΔP 3-tap [%]	ΔP 5-tap [%]
PE HC	+0.59	−0.54	− 4.43	+0.76
PE HC App	+1.81	+4.08	+1.23	+12.19
PE LF	+1.81	+3.26	+1.14	+3.63
PE LF App	+0.59	+3.85	+1.04	+7.32

Table 4.8: Luma Approximate relative percentage comparisons with the original design

Conclusion

Besides the improvements introduced by the HEVC project in compression efficiency with respect to the previous standards, software solutions show limitations in real-time encoding and decoding. The interpolations filters of this system represent the bottleneck for the CPU time: therefore an hardware implementation is strictly necessary in order to fulfill a short time coding requirement.
The starting architecture introduces several optimizations, namely, i) the amount of memory have been halved, ii) multipliers have been substituted with sums and shifts. The optimized multiplier-less two-dimensional filter architecture exploits hardware reconfiguration, throughput adaption and clock gating, offering a good energy-quality trade-off. Even if this final solution is able to provide higher throughput than the original software architecture, some improvements can be applied at different levels of the design: thus an appropriate internal architecture for the adders that are involved in the filtering operation is proposed.
In the two Legacy architectures, a first data analysis is carried out in order to apply both exact and approximate architectures. Several models for the adder configurations are applied: after a performance and precision evaluation, the best solution is chosen. Concerning the Chroma Legacy architecture, parallel & prefix correct adders configuration that employs a Ladner-Fischer prefix-processing stage results the most convenient choice in terms of speed enhancement, at the cost of a negligible area overhead and power dissipation. As regards the Luma Legacy architecture, the adoption of an approximate solution involving Generic Accuracy Configurable Adders is proposed: the effect of this inexact solution in terms of PSNR degradation on the entire HEVC system is evaluated. Results show that the solution with approximate encoder and decoder appears the best choice for an acceptable PSNR as output of the entire system.
An approximate DCT-IF stating architecture has been introduced in order to reduce the computational complexity of interpolation filters, by reaching a good trade-off between a good quality of video-coding and energy consumption, by dynamically reducing the number of taps. Indeed a lower order for the filters consists both in a reduced power dissipation and in a higher maximum frequency, with the drawback of a lower precision in the data out of the interpolation process. For the Luma Approximate structure, a slight speed improvement with a reduction in power and occupied area is earned through the employment of Han-Carlson correct adders, while for the Chroma Approximate architecture the starting structure is maintained.
Further improvements are still achievable in the proposed architecture: for example pipelining can be applied to further improve the system performances achieving a higher number of fractional pixels per second.
The implemented architectures are fully standard compliant, addressing the 1-D and 2-D interpolation process of all the different luma and chroma prediction unit sizes adopted by HEVC.

Appendix A

Parallel & Prefix Adders

Han Carlson Exact Architecture

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity Han_Carlson_Correct_Adder_Nbits is
     generic(N:positive:=22);
     port(
          In1, In2             :IN std_logic_vector(N-1 downto 0)
             ;      -- adder inputs
          out_A                      :OUT std_logic_vector(N-1
             downto 0)  -- adder output
     );
end entity Han_Carlson_Correct_Adder_Nbits;

architecture structural of Han_Carlson_Correct_Adder_Nbits is

component GP_Unit is
     port(
          A,B                  :IN std_logic;
          G,P                  :OUT std_logic
     );
end component;

component ParalPrefix_Unit is
     port(
          G1,G0,P1,P0          :IN std_logic;
          G01,P01              :OUT std_logic
     );
end component ParalPrefix_Unit;

signal gen_bits, prop_bits     : std_logic_vector(N-1 DOWNTO 0);
signal G_s1, P_s1                   : std_logic_vector(N/2-1
   downto 0);
```

```
30 signal G_s2, P_s2                    : std_logic_vector(N/2-2
   downto 0);
31 signal G_s3, P_s3                    : std_logic_vector(N/2-3
   downto 0);
32 signal G_s4, P_s4                    : std_logic_vector(N/2-5
   downto 0);
33 signal G_s5, P_s5                    : std_logic_vector(N/2-9
   downto 0);
34 signal G_s6, P_s6                    : std_logic_vector(N-1 downto
    0);
35 signal c_i_min1                      : std_logic_vector(N-1 downto
    0);
36
37 begin
38
39      -- BLOCK 1: gi,pi GENERATION
40      gp_gen: for i in N-1 downto 0 generate
41                GP_U: GP_Unit port map(A=>In1(i), B=>In2(i), G=>
                     gen_bits(i), P=>prop_bits(i));
42      end generate gp_gen;
43
44      ------------------------------------------------
45      --BLOCK 2: parallel & prefix problem with Han-Carlson
           Architecture
46
47      --outer row: Brent&Kung architecture
48      PP0_s1: for i in 1 to N/2 generate
49                PP0_U1: ParalPrefix_Unit port map(G1=>gen_bits(2*
                     i-1),G0=>gen_bits(2*i-2),P1=>prop_bits(2*i-1),
                     P0=>prop_bits(2*i-2),G01=>G_s1(i-1),P01=>P_s1(i
                     -1));
50      end generate PP0_s1;
51
52      --inner rows: Kogge-Stone architecture
53      PP0_s2: for i in 1 to N/2-1 generate
54                PP0_U2: ParalPrefix_Unit port map(G1=>G_s1(i),G0
                     =>G_s1(i-1),P1=>P_s1(i),P0=>P_s1(i-1),G01=>G_s2
                     (i-1),P01=>P_s2(i-1));
55      end generate PP0_s2;
56
57      PP0_S3: for i in 1 to N/2-2 generate
58
59                middleBit_s3: if i>1 AND i<=N/2-2 generate
60                PP0_U3: ParalPrefix_Unit port map(G1=>G_s2(i),G0
                     =>G_s2(i-2),P1=>P_s2(i),P0=>P_s2(i-2),G01=>G_s3
                     (i-1),P01=>P_s3(i-1));
61                END generate middleBit_s3;
62
```

```
63                LSBs_s3: if i=1 generate
64                PP0_U3: ParalPrefix_Unit port map(G1=>G_s2(i),G0
                     =>G_s1(i-1),P1=>P_s2(i),P0=>P_s1(i-1),G01=>G_s3
                     (i-1),P01=>P_s3(i-1));
65                END generate LSBs_s3;
66
67        end generate PP0_s3;
68
69
70        PP0_s4: for i in 1 to N/2-4 generate
71
72                middleBit_s4: if i>2 AND i<=N/2-4 generate
73                PP0_U4: ParalPrefix_Unit port map(G1=>G_s3(i+1),
                     G0=>G_s3(i-3),P1=>P_s3(i+1),P0=>P_s3(i-3),G01=>
                     G_s4(i-1),P01=>P_s4(i-1));
74                END generate middleBit_s4;
75
76                LSB1_s4: if i=2 generate
77                PP0_U4: ParalPrefix_Unit port map(G1=>G_s3(i+1),
                     G0=>G_s2(i-2),P1=>P_s3(i+1),P0=>P_s2(i-2),G01=>
                     G_s4(i-1),P01=>P_s4(i-1));
78                END generate LSB1_s4;
79
80                LSB0_s4: if i=1 generate
81                PP0_U4: ParalPrefix_Unit port map(G1=>G_s3(i+1),
                     G0=>G_s1(i-1),P1=>P_s3(i+1),P0=>P_s1(i-1),G01=>
                     G_s4(i-1),P01=>P_s4(i-1));
82                END generate LSB0_s4;
83
84        end generate PP0_s4;
85
86                --last row Kogge-Stone to be pruned for the
                     approximate version
87        PP0_s5: for i in 1 to N/2-8 generate
88
89                middleBit_s5: if i>=3 AND i<=N/2-8 generate
90                PP0_U5: ParalPrefix_Unit port map(G1=>G_s4(i+3),
                     G0=>G_s3(i-3),P1=>P_s4(i+3),P0=>P_s3(i-3),G01=>
                     G_s5(i-1),P01=>P_s5(i-1));
91                END generate middleBit_s5;
92
93                LSB1_s5: if i=2 generate
94                PP0_U5: ParalPrefix_Unit port map(G1=>G_s4(i+3),
                     G0=>G_s2(i-2),P1=>P_s4(i+3),P0=>P_s2(i-2),G01=>
                     G_s5(i-1),P01=>P_s5(i-1));
95                END generate LSB1_s5;
96
97                LSB0_s5: if i=1 generate
```

```
98                PP0_U5: ParalPrefix_Unit port map(G1=>G_s4(i+3),
                     G0=>G_s1(i-1),P1=>P_s4(i+3),P0=>P_s1(i-1),G01=>
                     G_s5(i-1),P01=>P_s5(i-1));
99                END generate LSB0_s5;
100
101       end generate PP0_s5;
102
103       --final row: Brent&Kung architecture
104
105       PP0_s6: for i in 1 to N/2 generate
106
107                MSB_odd: if i=N/2 generate
108                G_s6(2*i-1) <= G_s5(i-9);
109                P_s6(2*i-1) <= P_s5(i-9);
110                END generate;
111
112                MSB_s6: if i>8 AND i<=N/2-1 generate
113                PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                     i),G0=>G_s5(i-9),P1=>prop_bits(2*i),P0=>P_s5(i
                     -9),G01=>G_s6(2*i),P01=>P_s6(2*i));
114                G_s6(2*i-1) <= G_s5(i-9);
115                P_s6(2*i-1) <= P_s5(i-9);
116                END generate MSB_s6;
117
118                middle_s6_s4: if i>4 AND i<=8 generate
119                PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                     i),G0=>G_s4(i-5),P1=>prop_bits(2*i),P0=>P_s4(i
                     -5),G01=>G_s6(2*i),P01=>P_s6(2*i));
120                G_s6(2*i-1) <= G_s4(i-5);
121                P_s6(2*i-1) <= P_s4(i-5);
122                END generate middle_s6_s4;
123
124                middle_s6_s3: if i>2 AND i<=4 generate
125                PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                     i),G0=>G_s3(i-3),P1=>prop_bits(2*i),P0=>P_s3(i
                     -3),G01=>G_s6(2*i),P01=>P_s6(2*i));
126                G_s6(2*i-1) <= G_s3(i-3);
127                P_s6(2*i-1) <= P_s3(i-3);
128                END generate middle_s6_s3;
129
130                middle_s6_s2: if i=2 generate
131                PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                     i),G0=>G_s2(i-2),P1=>prop_bits(2*i),P0=>P_s2(i
                     -2),G01=>G_s6(2*i),P01=>P_s6(2*i));
132                G_s6(2*i-1) <= G_s2(i-2);
133                P_s6(2*i-1) <= P_s2(i-2);
134                END generate middle_s6_s2;
135
```

```
136                 middle_s6_s1: if i=1 generate
137                 PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                        i),G0=>G_s1(i-1),P1=>prop_bits(2*i),P0=>P_s1(i
                        -1),G01=>G_s6(2*i),P01=>P_s6(2*i));
138                 G_s6(2*i-1) <= G_s1(i-1);
139                 P_s6(2*i-1) <= P_s1(i-1);
140                 G_s6(2*i-2) <= gen_bits(i-1);
141                 P_s6(2*i-2) <= prop_bits(i-1);
142                 END generate middle_s6_s1;
143
144     end generate PP0_s6;
145
146     -------------------------------------
147     --BLOCK 3: Carry computation
148     c_i_min1(0) <= '0';
149     Cin_gen: for i in 1 to N-1 generate
150                 c_i_min1(i) <= G_s6(i-1) OR (P_s6(i-1) AND
                        c_i_min1(0)); --since c0=0 because we're just
                        considering additions
151     end generate Cin_gen;
152
153     -------------------------------------
154     --BLOCK 4: Sum computation
155     Sum_gen: for i in 0 to N-1 generate
156                 out_A(i) <= prop_bits(i) XOR c_i_min1(i);
157     end generate Sum_gen;
158
159
160 end structural;
```

./Appendix/vhdlfiles/Han_Carlson_Correct_Adder_Nbits.vhd

Ladner Fischer Exact Architecture

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  entity Ladner_Fisher_Nbit_correct is
5       generic(N:positive:=22);
6       port(
7            In1, In2             :IN std_logic_vector(N-1 downto 0)
                 ;      -- adder inputs
8            out_A                      :OUT std_logic_vector(N-1
                 downto 0)  -- adder output
9       );
10 end entity Ladner_Fisher_Nbit_correct;
11
12 architecture structural of Ladner_Fisher_Nbit_correct is
```

```
13
14 component GP_Unit is
15      port(
16           A,B                    :IN std_logic;
17           G,P                    :OUT std_logic
18      );
19 end component;
20
21 component ParalPrefix_Unit is
22      port(
23           G1,G0,P1,P0            :IN std_logic;
24           G01,P01                :OUT std_logic
25      );
26 end component ParalPrefix_Unit;
27
28 signal gen_bits, prop_bits     : std_logic_vector(N-1 DOWNTO 0);
29 signal G_s1, P_s1                   : std_logic_vector(N/2-1
      downto 0);
30 signal G_s2, P_s2                   : std_logic_vector(4 downto
      0);
31 signal G_s3, P_s3                   : std_logic_vector(4 downto
      0);
32 signal G_s4, P_s4                   : std_logic_vector(3 downto
      0);
33 signal G_s5, P_s5                   : std_logic_vector(N/2-9
      downto 0);
34 signal G_s6, P_s6                   : std_logic_vector(N-1 downto
       0);
35 signal c_i_min1                     : std_logic_vector(N-1 downto
       0);
36
37
38
39 begin
40
41      -- BLOCK 1: gi,pi GENERATION
42      gp_gen: for i in N-1 downto 0 generate
43                GP_U: GP_Unit port map(A=>In1(i), B=>In2(i), G=>
                     gen_bits(i), P=>prop_bits(i));
44      end generate gp_gen;
45
46      ------------------------------------------------
47      --BLOCK 2: parallel & prefix problem with Ladner-Fisher
           Architecture
48
49      -- first two rows: Brent&Kung architecture
50      PP0_s1: for i in 1 to N/2 generate
```

```
51              PP0_U1: ParalPrefix_Unit port map(G1=>gen_bits(2*
                   i-1),G0=>gen_bits(2*i-2),P1=>prop_bits(2*i-1),
                   P0=>prop_bits(2*i-2),G01=>G_s1(i-1),P01=>P_s1(i
                   -1));
52      end generate PP0_s1;
53
54      PP0_s2: for i in 0 to 4 generate
55
56              MSB_22_20_s2: if i=4 AND N/2>=10 generate
57                      PP0_U2: ParalPrefix_Unit port map(G1=>
                           G_s1(2*i+1),G0=>G_s1(2*i),P1=>P_s1(2*
                           i+1),P0=>P_s1(2*i),G01=>G_s2(i),P01=>
                           P_s2(i));
58              end generate MSB_22_20_s2;
59
60              MSB_18_s2: if i=4 AND N/2<10 generate
61                      G_s2(i) <= '0';
62                      P_s2(i) <= '0';
63              end generate MSB_18_s2;
64
65              middleBit_s2: if i<4 generate
66                      PP0_U2: ParalPrefix_Unit port map(G1=>
                           G_s1(2*i+1),G0=>G_s1(2*i),P1=>P_s1(2*
                           i+1),P0=>P_s1(2*i),G01=>G_s2(i),P01=>
                           P_s2(i));
67              end generate middleBit_s2;
68
69      end generate PP0_s2;
70
71      -- inner rows: Ladner-Fisher architecture
72
73      PP0_s3: for i in 0 to 4 generate
74
75              MSBs_s3: if i=4 AND N/2=11 generate
                                    -- for MSB of N/2=11 one
                   more operator is needed
76              PP0_U3: ParalPrefix_Unit port map(G1=>G_s1(2*i+2)
                   ,G0=>G_s2(i),P1=>P_s1(2*i+2),P0=>P_s2(i),G01=>
                   G_s3(i),P01=>P_s3(i));
77              END generate MSBs_s3;
78
79              MSB_18_s3: if i=4 AND N/2<11 generate
80                      G_s3(i) <= '0';
81                      P_s3(i) <= '0';
82              end generate MSB_18_s3;
83
84              middleBitEven_s3: if ((i mod 2) = 0) AND i<=3
                   generate           --for even numbers
```

```
85                PP0_U3: ParalPrefix_Unit port map(G1=>G_s1(2*i+2)
                     ,G0=>G_s2(i),P1=>P_s1(2*i+2),P0=>P_s2(i),G01=>
                     G_s3(i),P01=>P_s3(i));
86                END generate middleBitEven_s3;
87
88                middleBitOdd_s3: if ((i mod 2) /= 0) AND i<=3
                     generate                --for odd numbers
89                PP0_U3: ParalPrefix_Unit port map(G1=>G_s2(i),G0
                     =>G_s2(i-1),P1=>P_s2(i),P0=>P_s2(i-1),G01=>G_s3
                     (i),P01=>P_s3(i));
90                END generate middleBitOdd_s3;
91
92        end generate PP0_s3;
93
94
95        -- last two rows of Ladner-Fisher to be pruned in
             approximate version
96
97        PP0_s4: for i in 0 to 3 generate
98
99                middleBit_s4: if i>1 AND i<=3 generate
100               PP0_U4: ParalPrefix_Unit port map(G1=>G_s3(i),G0
                     =>G_s3(1),P1=>P_s3(i),P0=>P_s3(1),G01=>G_s4(i),
                     P01=>P_s4(i));
101               END generate middleBit_s4;
102
103               LSB1_s4: if i=1 generate
104               PP0_U4: ParalPrefix_Unit port map(G1=>G_s2(i+1),
                     G0=>G_s3(1),P1=>P_s2(i+1),P0=>P_s3(1),G01=>G_s4
                     (i),P01=>P_s4(i));
105               END generate LSB1_s4;
106
107               LSB0_s4: if i=0 generate
108               PP0_U4: ParalPrefix_Unit port map(G1=>G_s1(i+4),
                     G0=>G_s3(1),P1=>P_s1(i+4),P0=>P_s3(1),G01=>G_s4
                     (i),P01=>P_s4(i));
109               END generate LSB0_s4;
110
111       end generate PP0_s4;
112
113       PP0_s5: for i in 0 to N/2-9 generate
114
115               middleBit_s5: if i=2 generate
116               PP0_U5: ParalPrefix_Unit port map(G1=>G_s3(i+2),
                     G0=>G_s4(3),P1=>P_s3(i+2),P0=>P_s4(3),G01=>G_s5
                     (i),P01=>P_s5(i));
117               END generate middleBit_s5;
118
```

```
119                 LSB1_s5: if i=1 generate
120                 PP0_U5: ParalPrefix_Unit port map(G1=>G_s2(i+3),
                       G0=>G_s4(3),P1=>P_s2(i+3),P0=>P_s4(3),G01=>G_s5
                       (i),P01=>P_s5(i));
121                 END generate LSB1_s5;
122
123                 LSB0_s5: if i=0 generate
124                 PP0_U5: ParalPrefix_Unit port map(G1=>G_s1(8),G0
                       =>G_s4(3),P1=>P_s1(8),P0=>P_s4(3),G01=>G_s5(i),
                       P01=>P_s5(i));
125                 END generate LSB0_s5;
126
127     end generate PP0_s5;
128
129
130     --final row: Brent&Kung architecture
131
132     PP0_s6: for i in 1 to N/2 generate
133
134                 MSB_odd: if i=N/2 generate
135                 G_s6(2*i-1) <= G_s5(i-9);
136                 P_s6(2*i-1) <= P_s5(i-9);
137                 END generate;
138
139                 MSB_s6: if i>8 AND i<=N/2-1 generate
140                 PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                       i),G0=>G_s5(i-9),P1=>prop_bits(2*i),P0=>P_s5(i
                       -9),G01=>G_s6(2*i),P01=>P_s6(2*i));
141                 G_s6(2*i-1) <= G_s5(i-9);
142                 P_s6(2*i-1) <= P_s5(i-9);
143                 END generate MSB_s6;
144
145
146                 middle_s6_s4: if i>4 AND i<=8 generate
147                 PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                       i),G0=>G_s4(i-5),P1=>prop_bits(2*i),P0=>P_s4(i
                       -5),G01=>G_s6(2*i),P01=>P_s6(2*i));
148                 G_s6(2*i-1) <= G_s4(i-5);
149                 P_s6(2*i-1) <= P_s4(i-5);
150                 END generate middle_s6_s4;
151
152
153                 middle_s6_s3: if i>2 AND i<=4 generate
154                 PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                       i),G0=>G_s3(i-3),P1=>prop_bits(2*i),P0=>P_s3(i
                       -3),G01=>G_s6(2*i),P01=>P_s6(2*i));
155                 G_s6(2*i-1) <= G_s3(i-3);
156                 P_s6(2*i-1) <= P_s3(i-3);
```

```
157                 END generate middle_s6_s3;
158
159                 middle_s6_s2: if i=2 generate
160                 PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                       i),G0=>G_s2(i-2),P1=>prop_bits(2*i),P0=>P_s2(i
                       -2),G01=>G_s6(2*i),P01=>P_s6(2*i));
161                 G_s6(2*i-1) <= G_s2(i-2);
162                 P_s6(2*i-1) <= P_s2(i-2);
163                 END generate middle_s6_s2;
164
165                 middle_s6_s1: if i=1 generate
166                 PP0_U6: ParalPrefix_Unit port map(G1=>gen_bits(2*
                       i),G0=>G_s1(i-1),P1=>prop_bits(2*i),P0=>P_s1(i
                       -1),G01=>G_s6(2*i),P01=>P_s6(2*i));
167                 G_s6(2*i-1) <= G_s1(i-1);
168                 P_s6(2*i-1) <= P_s1(i-1);
169                 G_s6(2*i-2) <= gen_bits(i-1);
170                 P_s6(2*i-2) <= prop_bits(i-1);
171                 END generate middle_s6_s1;
172
173       end generate PP0_s6;
174
175       -------------------------------------
176       --BLOCK 3: Carry computation
177       c_i_min1(0) <= '0';
178       Cin_gen: for i in 1 to N-1 generate
179                 c_i_min1(i) <= G_s6(i-1); -- OR (P_s6(i-1) AND
                       c_i_min1(0)); -- I can neglect the or gate
                       since due to addition c0=0
180       end generate Cin_gen;
181
182       -------------------------------------
183       --BLOCK 4: Sum computation
184       Sum_gen: for i in 0 to N-1 generate
185                 out_A(i) <= prop_bits(i) XOR c_i_min1(i);
186       end generate Sum_gen;
187
188
189
190 end structural;
```

./Appendix/vhdlfiles/Ladner_Fisher_Correct_Adder_Nbits.vhd

Generate Propagate Unit

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all;
```

```
4
5  entity GP_Unit is
6       port(
7            A,B                    :IN std_logic;
8            G,P                    :OUT std_logic
9       );
10 end entity GP_Unit;
11
12 architecture structure of GP_Unit is
13
14 begin
15
16      G <= A AND B;
17      P <= A XOR B;
18
19 end structure;
```

./Appendix/vhdlfiles/GP_Unit.vhd

Parallel & Prefix Unit

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity ParalPrefix_Unit is
6       port(
7            G1,G0,P1,P0            :IN std_logic;
8            G01,P01                :OUT std_logic
9       );
10 end entity ParalPrefix_Unit;
11
12 architecture structure of ParalPrefix_Unit is
13
14 begin
15
16      G01 <= G1 OR (P1 AND G0);
17      P01 <= P0 AND P1;
18
19 end structure;
```

./Appendix/vhdlfiles/ParalPrefix_Unit.vhd

Appendix B

Generic Accuracy Configurable Adders

GeAr Adder k=3 P=0

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity GeAr_Adder_Nbits is
     generic(N:integer:=22; R:integer:=7; P:integer:=0); -- L=R+P, N=R+L
     port(
          In1, In2             :IN std_logic_vector(N-1 downto 0) ;     -- adder inputs
          c_in,c_in1                    :IN std_logic;
          c_out,c_out1         :OUT std_logic;
          out_A                         :OUT std_logic_vector(N-1 downto 0)  -- adder output
     );
end entity GeAr_Adder_Nbits;

architecture structural of GeAr_Adder_Nbits is

--signals
signal temp                                 : std_logic_vector(N-1 downto 2*R+P-1);
signal temp1                                : std_logic_vector(2*R+P downto R+P-1);  --it includes carry-in and carry-out to be used
signal sum1_MSB                                   : std_logic_vector(N-1 downto 2*R+P);
signal sum1_middle                          : std_logic_vector(2*R+P-1 downto R+P);
signal cg_o, cp_o,cg_o1, cp_o1              : std_logic;
signal ED,ED1                               : std_logic;
```

```
23 signal sum1_LSB                                  : std_logic_vector(
      R+P downto 0); --it includes carry-out to be used
24 begin
25
26 ----BLOCK 1: MSBs------------------------
27 --SUM for Most Significant Bits
28 temp <= std_logic_vector(signed(In1(N-1 downto 2*R+P)&'1')+
      signed(In2(N-1 downto 2*R+P)&cg_o));
29 sum1_MSB <= temp(N-1 downto 2*R+P);
30
31 --carry generation for Most Significant block with CG Unit
32 cg_o <= (In1(2*R+P-1) and In2(2*R+P-1)) or (c_in and (In1(2*R+P
      -1) xor In2(2*R+P-1)));
33 cp_o <= In1(2*R+P-1) xor In2(2*R+P-1);
34
35 --Error detection for c_in unit
36 ED <= (temp1(2*R+P) xor c_in) AND cp_o;
37 c_out <= c_in xor ED;
38
39 -----BLOCK 2: middle bits---------------------
40 --SUM for intermediate Bits
41 temp1 <= std_logic_vector(signed('0'&In1(2*R+P-1 downto R+P)
      &'1')+signed('0'&In2(2*R+P-1 downto R+P)&cg_o1)); --temp1(2R+P
      )=carry out
42 sum1_middle <= temp1(2*R+P-1 downto R+P);
43
44 --carry generation for Most Significant block with CG Unit
45 cg_o1 <= (In1(R+P-1) and In2(R+P-1)) or (c_in1 and (In1(R+P-1)
      xor In2(R+P-1)));
46 cp_o1 <= In1(R+P-1) xor In2(R+P-1);
47
48 --Error detection for c_in unit
49 ED1 <= (sum1_LSB(R+P) xor c_in1) AND cp_o1;
50 c_out1 <= c_in1 xor ED1;
51
52 ----BLOCK 3: LSBs------------------------
53 --SUM for Least Significant Bits
54 sum1_LSB <= std_logic_vector(signed('0'&In1(R+P-1 downto 0)) +
      signed('0'&In2(R+P-1 downto 0))); --sum1_LSB(R+P)=c_out,
      sum1_LSB(R+P-1:0)=out_A(R+P-1:0)
55
56 --sum out is composed of outcomes of the three sub-blocks
57 out_A <= sum1_MSB(N-1 downto 2*R+P)&sum1_middle(2*R+P-1 downto R
      +P)&sum1_LSB(R+P-1 downto 0);
58
59 end structural;
```

./Appendix/vhdlfiles/GeAr_Adder_Nbits_P0_3Blocks.vhd

Bibliography

[1] K. Sayood. *Introduction to Data Compression - Third Edition*, pp 571-614. San-Francisco, CA, USA:Morgan Kauffman Publishers Inc., 2005. 2

[2] A.Giannini. *Complexity analysis and optimized interpolation filter hardware architecture for HEVC*. Turin, Dec 2017. 4, 5, 6, 8, 9, 11, 12, 13, 14, 15, 54, 57, 58, 59, 60, 61, 62

[3] G.J.Sullivan, J-R Ohm, W-J.Han, T. Wiegand. *Overview of the High Efficiency Video Coding (HEVC) Standard*. IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, pp. 1649-1668, Dec 2012. 6, 7

[4] F. Bossen, B. Bross, K. Suhring, D. Flynn. *HEVC Complexity and Implementation Analysis*. IEEE Transactions on Circuits and Systems for Video Technology,vol.22, pp. 1685-1696, Dec 2016.

[5] G. Pastuszak. *High-Speed Architecture of the CABAC Probability Modeling for H.265/HEVC Encoders*. In 2016 International Conference On Signals and Electronic Systems (ICSES), Sept 2016.

[6] ITU-T Video Coding Experts Group and ISO/IEC Moving Picture Experts Group. *HM16.15*. `https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-16.15/` 5, 6, 22, 48, 56

[7] A. Giannini *Interpolation Filters*. `https://github.com/Jak94/InterpolationFilters` 8

[8] C. M. Diniz, M. Shafique, S. Bampi, J. Henkel *A reconfigurable hardware architecture for fractional pixel interpolation in high efficiency video coding*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 34, pp. 238-251, Feb 2015. 10

[9] J. R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand *Comparison of the coding efficiency of video coding standards - including high efficiency video coding (hevc)*. IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, pp. 1669-1684, Dec. 2012. 22

[10] M. Macedo, L. Soares, B. Silveira, C. M. Diniz, E. A. C. da Costa *Exploring the Use of Parallel Prefix Adder Topologies into Approximate Adder Circuits*. IEEE Transactions on Circuits and Systems, pp. 298-301, 2017. 24

[11] D. Esposito, D. De Caro, E. Napoli, N. Petra, A. G. M. Strollo *Variable Latency Speculative Han-Carlson Adder*. IEEE Transactions on Circuits and Systems, vol. 62, pp. 1353-1359, May 2015. 26

[12] D. Esposito, D. De Caro, A. G. M. Strollo *Variable Latency Speculative Parallel Prefix Adders for Unsigned and Signed Operands.* IEEE Transactions on Circuits and Systems, vol. 63, pp. 1200-1209, Aug 2016. 27, 28, 33

[13] S. Mazahir, O. Hasan, M. Shafique *Adaptive Approximate Computing in Arithmetic Dat-apaths.* IEEE Design and Test, pp. 1-8, 2017. 37

[14] M. Shafique, W. Ahmad, R: Hafiz, J. Henkel *A low latency generic accuracy configurable adder.* in Proc 52nd Annual Des. Autom. Conf., p.86, 2015. 38

[15] G. Casella, R. L. Berger *Statistical Inference (2nd ed.)* p.102. Duxbury Advanced Series, 2001. 45

[16] E. Nogues, D. Menard, and M. Pelcat *Algorithmic-level approximate computing applied to energy efficient hevc decoding* IEEE Transactions on Emerging Topics in Computing, vol PP.,no 99, pp. 1-12, 2016

[17] F. Bossen *Common test conditions and software reference configurations* Joint Collaborative Team on Video Coding (JCT-VC) of ITU-T SG 16 Wp 3 and ISO/IEC JTC 1/SC 29/WG 11, Geneva, January 2013. 12th meeting.

23, 48, 49, 50, 51, 52, 56